Appunti di Algoritmi e Strutture Dati

Mauro Tempesta

10 marzo 2011

Indice

Capitolo 1

Introduzione informale agli algoritmi

Definizione 1.1. Un *algoritmo* è un insieme di istruzioni, definite passo per passo, in modo tale da poter essere eseguite meccanicamente, e tali da produrre un determinato risultato.

1.1 I numeri di Fibonacci

Si vuole scrivere un algoritmo per il calcolo dell'n-esimo numero di Fibonacci, che può essere calcolato utilizzando la seguente formula:

$$F_n = \begin{cases} 1 & \text{se } n = 1, 2 \\ F_{n-1} + F_{n-2} & \text{se } n \geq 3 \end{cases} \tag{1.1}$$

Saranno presentati diversi algoritmi, evidenziandone pregi e difetti.

1.1.1 Algoritmo numerico

Un primo algoritmo si basa sull'utilizzo di una funzione matematica che calcoli direttamente i numeri di Fibonacci; proviamo a vedere se è possibile individuarne una esponenziale della forma a^n con $a \neq 0$ che soddisfi la relazione di ricorrenza 1.1:

$$a^n = a^{n-1} + a^{n-2}$$
$$a^{n-2} \cdot (a^2 - a - 1) = 0$$

Poiché, per ipotesi, $a \neq 0$, cerchiamo i valori di a che soddisfano l'equazione:

$$a^2 - a - 1 = 0 \tag{1.2}$$

L'equazione 1.2 ammette due radici reali:

$$\phi = \frac{1 + \sqrt{5}}{2} \approx +1.618$$
$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618$$

Le funzioni ϕ^n e $\hat{\phi}^n$ soddisfano entrambe la relazione (1.1), ma nessuna di esse calcola correttamente i numeri di Fibonacci come vorremmo: ad esempio, $\phi^2 \neq F_2$; questo perché non sono stati considerati i passi base della definizione ricorsiva.
Per risolvere il problema, è sufficiente osservare che una qualunque combinazione lineare di funzioni che soddisfano la relazione di Fibonacci soddisfa anch'essa tale relazione; cerchiamo opportune costanti c_1 e c_2 che diano la funzione cercata:

$$\begin{cases} c_1 \cdot \phi + c_2 \cdot \hat{\phi} = 1 \\ c_1 \cdot \phi^2 + c_2 \cdot \hat{\phi}^2 = 1 \end{cases}$$

Risolvendo il sistema si ottiene:

$$c_1 = +\frac{1}{\sqrt{5}} \quad c_2 = -\frac{1}{\sqrt{5}}$$

Segue che il primo algoritmo per il calcolo dei numeri di Fibonacci è il seguente:

algoritmo `fibonacciNumerico` (*intero* n) $\rightarrow$ *intero*

```
1 return  1/√5 · (φ^n − φ̂^n)
```

Il limite di tale algoritmo è dato dal fatto che si è costretti ad operare con numeri reali, rappresentati nei calcolatori con precisione limitata, e quindi si possono fornire risposte errate dovute ad errori di arrotondamento.

1.1.2 Algoritmo ricorsivo

Data la natura ricorsiva della relazione di Fibonacci, si può pensare di realizzare un algoritmo ricorsivo, come il seguente:

algoritmo `fibonacciRicorsivo` (*intero* n) $\rightarrow$ *intero*

```
1 if(n <= 2) then
2   return 1
3 else
4   return fibonacciRicorsivo(n - 1) + fibonacciRicorsivo(n - 2)
```

Per analizzare le prestazioni di un algoritmo, si possono considerare il numero di linee di codice eseguite (*complessità temporale*) e la quantità di memoria occupata (*complessità spaziale*): consideriamo la prima. In generale, ogni algoritmo ricorsivo può essere analizzato mediante una *relazione di ricorrenza*: il tempo speso da una routine è pari al tempo speso all'interno della routine più quello speso dalle chiamate ricorsive; ad esempio, la relazione per l'algoritmo sopra descritto, per $n > 2$, è la seguente:

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

Possiamo rappresentare le chiamate ricorsive con una struttura ad albero, detta *albero di ricorsione*: si usa un nodo, la radice dell'albero, per la prima chiamata e generiamo un figlio per ogni chiamata ricorsiva.

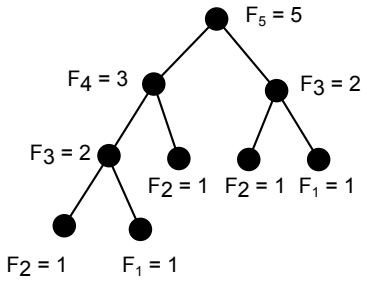


Figura 1.1: Albero di ricorsione di `fibonacciRicorsivo` per il calcolo di F_5

Per calcolare il numero di linee di codice eseguite da una generica chiamata `fibonacciRicorsivo`(n) usiamo i seguenti lemmi.

Lemma 1.1. *Sia T_n l'albero delle chiamate ricorsive della funzione `fibonacciRicorsivo`(n): il numero di foglie in T_n è pari al numero di Fibonacci F_n.*

Dimostrazione. Procediamo per induzione su n.

Caso base: è banalmente verificato; per $n = 1$, T_1 contiene un solo nodo, e dunque una sola foglia ($F_1 = 1$) e, per $n = 2$, T_2 contiene anch'esso un solo nodo, e quindi una sola foglia ($F_2 = 1$).

Ipotesi induttiva: sia $n > 2$ e supponiamo che il lemma sia verificato per ogni k tale per cui $2 \leq k \leq n - 1$.

Passo induttivo: usando l'ipotesi, dimostriamo che il lemma vale per n. L'albero della ricorsione T_n ha come sottoalbero sinistro T_{n-1} e come sottoalbero destro T_{n-2}: per ipotesi essi hanno, rispettivamente, F_{n-1} e F_{n-2} foglie, dunque T_n ha $F_{n-1} + F_{n-2} = F_n$ foglie, come si voleva dimostrare. □

Lemma 1.2. *Sia T un albero binario in cui ogni nodo interno ha esattamente due figli: allora il numero di nodi interni di T è pari al numero di foglie diminuito di uno.*

Dimostrazione. Procediamo per induzione su n.

Caso base: se $n = 1$, T ha una sola foglia e nessun nodo interno, quindi la condizione è verificata.

Ipotesi induttiva: supponiamo per ipotesi che la condizione valga per tutti gli alberi con meno di n nodi.

Passo induttivo: proviamo che la condizione valga anche per T, ossia che $i = f - 1$, dove i è il numero di nodi interni di T e f il numero di foglie di T; sia $\hat{T}$ un albero ottenuto da T rimuovendo una qualunque coppia di foglie aventi lo stesso padre: $\hat{T}$ avrà $i-1$ nodi interni e $f-2+1 = f-1$ foglie. Per ipotesi, poiché $\hat{T}$ ha meno nodi di T, vale la relazione $i - 1 = (f - 1) - 1$: sommando uno ad ambo i membri, si ottiene $i = f - 1$, ossia l'uguaglianza che si voleva dimostrare. □

Alla luce di quanto appena dimostrato, la chiamata generica `fibonacciRicorsivo`(n) comporta l'esecuzione di F_n righe di codice per via delle foglie (una per ciascuna foglia) e $2 \cdot (F_n - 1)$ righe per via dei nodi interni (due per ciascuno), per un totale di $3 \cdot F_n - 2$ righe di codice, una soluzione assai inefficiente.

1.1.3 Algoritmo iterativo

La lentezza di `fibonacciRicorsivo` è dovuta al fatto che continua a ricalcolare ripetutamente la soluzione dello stesso sottoproblema (vedi, ad esempio, F_3 nell'albero di ricorsione in figura 1.1); per fare di meglio, si potrebbe risolvere il sottoproblema una volta sola, memorizzarne la soluzione ed usarla nel seguito invece di ricalcolarla: questa è l'idea che sta alla base della tecnica chiamata *programmazione dinamica*.

algoritmo `fibonacciIterativo` (*intero* n) → *intero*

```
1 Fib: array interi
2 Fib[1] = Fib[2] = 1
3 for i = 3 to n do
4   Fib[i] = Fib[i - 1] + Fib[i - 2]
5 return Fib[n]
```

Per quanto riguarda l'analisi temporale, occorre operare in maniera diversa: per ogni linea di codice, calcoliamo quante volte essa è eseguita, esaminando a quali cicli appartiene e quante volte essi sono eseguiti. Nel nostro caso si ha:

$$T(n) = \begin{cases} 4 & \text{se } n = 1, 2 \\ 2n & \text{se } n > 2 \end{cases}$$

Si tratta di una soluzione decisamente più efficiente dell'algoritmo ricorsivo proposto.

Un piccolo miglioramento

L'algoritmo `fibonacciIterativo` richiede una quantità di spazio di memoria linearmente proporzionale alla dimensione dell'input n, anche se ogni iterazione utilizza solo i due valori precedenti a F_n, F_{n-1} e F_{n-2}; rimpiazzando l'array con due variabili, come proposto nel seguente algoritmo, otteniamo un notevole risparmio di memoria.

algoritmo `fibonacciIterativoModificato` (*intero* n) → *intero*

```
1 a = 1, b = 1
2 for i = 3 to n do
3   c = a + b
4   a = b
5   b = c
6 return b
```

1.1.4 Algoritmo basato su potenze ricorsive

L'algoritmo che sarà presentato in questa sezione si basa sul seguente lemma.

Lemma 1.3. *Sia* $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. *Allora* $A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$.

Dimostrazione. Procediamo per induzione su n.

Caso base: sia $F_0 = 0$ fissato per convenzione. Per $n = 2$ il caso base è banalmente verificato:
$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$

Ipotesi induttiva: supponiamo valido il lemma per $n-1$, ossia:
$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix}$

Passo induttivo: dimostriamo la validità del lemma per n:
$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{n-1} + F_{n-2} & F_{n-1} \\ F_{n-2} + F_{n-3} & F_{n-2} \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \quad \square$$

Usando il lemma appena dimostrato ed utilizzando il metodo dei quadrati ripetuti per il calcolo della potenza n-esima della matrice, si ottiene il seguente algoritmo:

algoritmo `fibonacciMatrice` (*intero* n) $\rightarrow$ *intero*

```
1  M = ( 1 1 )
       ( 1 0 )
2  potenzaDiMatrice(M, n - 1)
3  return M[0][0]

4  potenzaDiMatrice(matrice M, intero n)
5  if(n > 1) then
6    potenzaDiMatrice(M, n / 2)
7    M = M * M
8    if(n dispari)
9      M = M * ( 1 1 )
               ( 1 0 )
```

L'algoritmo presenta la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} O(1) & \text{se } n \leq 1 \\ T(n/2) + O(1) & \text{se } n > 1 \end{cases}$$

la cui soluzione è:

$$T(n) = O(\log n)$$

Capitolo 2

Modelli di calcolo e metodologie di analisi

2.1 Criteri di costo

Il criterio di misurazione del tempo di esecuzione di un algoritmo che si basa sull'assunzione che le diverse operazione richiedano tutte lo stesso tempo, indipendentemente dalla dimensione degli operandi coinvolti, è noto come *misura di costo uniforme*; si tratta di un criterio utile in prima approssimazione, ma si tratta di un modello troppo idealizzato.
Per ovviare a questo problema, è stato proposto un criterio, noto come *misura di costo logaritmico*, che assume che il costo di esecuzione delle operazioni dipenda dalla dimensione degli operandi coinvolti; nonostante fornisca una buona approssimazione, in svariati casi può generare una complessità eccessiva e non necessaria.

2.2 La notazione asintotica

Definizione 2.1. Data una funzione $f(n)$, definiamo:

- $O(f(n)) = \{g(n) : \exists c > 0 \wedge n_0 \geq 0 : g(n) \leq cf(n), \forall n \geq n_0\}$ ($g(n)$ cresce *al più* come $f(n)$);
- $\Omega(f(n)) = \{g(n) : \exists c > 0 \wedge n_0 \geq 0 : g(n) \geq cf(n), \forall n \geq n_0\}$ ($g(n)$ cresce *almeno* come $f(n)$);
- $\Theta(f(n)) = \{g(n) : \exists c_1, c_2 > 0 \wedge n_0 \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$ ($g(n)$ cresce *esattamente* come $f(n)$).

Proprietà 2.1. Date due funzioni $f(n)$ e $g(n)$, risulta $g(n) = \Theta(f(n))$ se e solo se $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$.

Proprietà 2.2. Θ gode della proprietà simmetrica: $g(n) = \Theta(f(n))$ se e solo se $f(n) = \Theta(g(n))$.

Proprietà 2.3. O, Ω sono simmetriche trasposte: $f(n) = \Omega(g(n))$ se e solo se $g(n) = O(f(n))$.

Proprietà 2.4. Per tutte e tre le notazioni vale la proprietà transitiva: per la notazione O, ad esempio, $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$.

Teorema 2.1. *Siano $f(n)$ e $g(n)$ funzioni positive:*

- $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = +\infty$
  $\to \exists M > 0 \wedge n_0 > 0 : \frac{f(n)}{g(n)} \geq M, \forall n \geq n_0$
  $\to f(n) \geq M \cdot g(n)$
  $\to f(n) = \Omega(g(n)) \wedge g(n) = O(f(n))$
- $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$
  $\to$ *preso* $\epsilon > 0, \exists n_0 > 0 : \frac{f(n)}{g(n)} \leq \epsilon, \forall n \geq n_0$
  $\to f(n) \leq \epsilon \cdot g(n)$
  $\to f(n) = O(g(n)) \wedge g(n) = \Omega(f(n))$

- $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = l \neq 0$
  $\to$ *preso* $\epsilon > 0, \exists n_0 > 0 : \left|\frac{f(n)}{g(n)} - l\right| \leq \epsilon, \forall n \geq n_0$
  $\to -\epsilon \leq \frac{f(n)}{g(n)} - l \leq \epsilon$
  $\to l - \epsilon \leq \frac{f(n)}{g(n)} \leq l + \epsilon$
  $\to (l - \epsilon) \cdot g(n) \leq f(n) \leq (l + \epsilon) \cdot g(n)$
  $\to f(n) = \Theta(g(n))$

2.3 Delimitazioni inferiori e superiori

Definizione 2.2. Un algoritmo A ha costo di esecuzione $O(f(n))$ su istanze di ingresso di dimensione n e rispetto ad una certa risorsa di calcolo, se la quantità r di risorsa *sufficiente* per eseguire A su una qualunque istanza di dimensione n verifica la relazione $r(n) = O(f(n))$.

Definizione 2.3. Un problema P ha complessità $O(f(n))$ rispetto ad una data risorsa di calcolo se *esiste* un algoritmo che risolve P il cui costo di esecuzione rispetto a quella risorsa è $O(f(n))$.

Definizione 2.4. Un algoritmo A ha costo di esecuzione $\Omega(f(n))$ su istanze di dimensione n e rispetto ad una certa risorsa di calcolo, se la massima quantità r di risorsa *necessaria* per eseguire A su istanze di dimensione n verifica la relazione $r(n) = \Omega(f(n))$.

Definizione 2.5. Un problema P ha complessità $\Omega(f(n))$ rispetto ad una data risorsa di calcolo se *ogni* algoritmo che risolve P ha costo di esecuzione $\Omega(f(n))$ rispetto a quella risorsa.

Definizione 2.6. Dato un problema P con complessità $\Omega(f(n))$ rispetto ad una data risorsa di calcolo, un algoritmo che risolve P è *ottimo* se ha costo di esecuzione $O(f(n))$ rispetto a quella risorsa.

2.4 Metodi di analisi

Per analizzare il tempo di esecuzione di un algoritmo, solitamente si usa distinguere fra tre diverse categorie di istanze, a parità di dimensione; sia $T(I)$ il tempo di esecuzione dell'algoritmo sull'istanza I:

Caso peggiore:

$$T_{worst}(n) = \max_{\substack{\text{istanze di} \\ \text{dimensione } n}} T(I)$$

Caso migliore:

$$T_{best}(n) = \min_{\substack{\text{istanze di} \\ \text{dimensione } n}} T(I)$$

Caso medio: sia $P(I)$ la probabilità di occorrere dell'istanza I:

$$T_{avg}(n) = \sum_{\substack{\text{istanze di} \\ \text{dimensione } n}} (T(I) \cdot P(I))$$

2.5 Analisi di algoritmi ricorsivi

2.5.1 Metodo di iterazione

L'idea è quella di ridurre la ricorsione ad una sommatoria dipendente solo dalla dimensione del problema iniziale.

Esempio 2.1. Sia data la seguente relazione di ricorrenza e assumiamo, per semplicità di analisi, che n sia una potenza di 3:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 9 \cdot T(n/3) + n & \text{se } n > 1 \end{cases}$$

Srotolando la ricorsione otteniamo:

$$\begin{aligned} T(n) &= 9 \cdot T(n/3) + n \\ &= 9 \cdot (9 \cdot T(n/9) + n/3) + n \\ &= 9^2 \cdot T(n/3^2) + 9 \cdot n/3 + n \\ &= ... \\ &= 9^i \cdot T(n/3^i) + \sum_{j=0}^{i-1} (9/3)^j n \end{aligned}$$

Dal momento che $n/3^i = 1$ quando $i = \log_3 n$, risulta:

$$T(n) = 9^{log_3 n} + n \cdot \sum_{j=0}^{\log_3 n - 1} 3^j$$

Poiché $\log_3 n = \log_9 n \cdot \log_3 9$, risulta $9^{\log_3 n} = 9^{\log_9 n \cdot \log_3 9} = n^2$; usando la serie geometrica, si ottiene:

$$T(n) = n^2 + \frac{3^{\log_3 n} - 1}{3 - 1} = n^2 + \frac{n-1}{2} = \Theta(n^2)$$

2.5.2 Metodo di sostituzione

L'idea è di intuire la soluzione della relazione di ricorrenza ed utilizzare il principio di induzione per dimostrare che l'intuizione è corretta.

Esempio 2.2. Sia data la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(\lfloor n/2 \rfloor) + n & \text{se } n > 1 \end{cases}$$

Intuizione: $T(n) = O(n)$, ossia dimostrare che esistono $c > 0$ e $n_0 > 0$ tali che $T(n) \leq c \cdot n, \forall n \geq n_0$

Passo base: provo $n = 1$
$T(1) = 1 \leq c \cdot 1 \rightarrow c \geq 1$

Ipotesi induttiva: supponiamo di avere $c > 0$ tale per cui $T(\lfloor n/2 \rfloor) \leq c \cdot \lfloor n/2 \rfloor$

Passo induttivo: sia $n > 1$

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

Rimane da provare che $c \cdot \frac{n}{2} + n \leq c \cdot n$; ciò risulta essere vero per $c \geq 2$.

Affinché siano verificati sia il caso base che il passo induttivo, occorre prendere una qualsiasi $c \geq 2$.

Esempio 2.3. Per illustrare altre sottigliezze relative all'uso del metodo di sostituzione, consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n = 1, 2 \\ 9 \cdot T(\lfloor n/3 \rfloor) + n & \text{se } n > 2 \end{cases}$$

Intuizione: $T(n) = O(n^2)$, ossia dimostrare che esistono $c > 0$ e $n_0 > 0$ tali che $T(n) \leq c \cdot n^2, \forall n \geq n_0$

Passo base: $T(1) = T(2) = 1 \leq c \cdot 1^2 \leq c \cdot 2^2 \ \rightarrow c \geq 1$

Ipotesi induttiva: assumiamo che la disuguaglianza sia soddisfatta per ogni $k < n$

Passo induttivo: sia $n > 2$

$$\begin{aligned} T(n) &= 9 \cdot T(\lfloor n/3 \rfloor) + n \\ &\leq 9 \cdot c \cdot \lfloor n/3 \rfloor^2 + n \\ &\leq 9 \cdot c \cdot (n/3)^2 + n \\ &= c \cdot n^2 + n \end{aligned}$$

Usando questa soluzione, non riusciamo a dimostrare la nostra affermazione poiché $c \cdot n^2 + n > c \cdot n^2$; possiamo risolvere facilmente il problema usando un'ipotesi induttiva più forte, in modo da far comparire un addendo negativo negativo che, sommato ad n, faccia tornare i conti; provando con l'ipotesi $T(n) \leq c \cdot (n^2 - n)$ otteniamo:

$$\begin{aligned} T(n) &= 9 \cdot T(\lfloor n/3 \rfloor) + n \\ &\leq 9 \cdot c \cdot ((n/3)^2 - n/3) + n \\ &= c \cdot n^2 - 3 \cdot c \cdot n + n \end{aligned}$$

Rimane da provare che $c \cdot n^2 - 3 \cdot c \cdot n + n \leq c \cdot (n^2 - n)$; ciò risulta essere vero per $c \geq 1/2$. Abbiamo, però, un altro problema: il passo base non è più verificato; infatti $T(1) = 1 > c \cdot (1^2 - 1) = 0$; ciò può essere risolto cambiando n_0, che prima avevamo scelto pari a 1.
Poiché $T(2) = 1 \leq c \cdot (4-2)$ per $c \geq 1/2$, vediamo se possiamo scegliere $n_0 = 2$: per poterlo fare, dobbiamo definire tutti i casi che si riconducono a $T(1)$ come casi base e verificare che, per tutti questi, esista c che verifichi la nostra condizione; in particolare, in questo esempio, dobbiamo definire come passi base $T(3), T(4), T(5)$, oltre a $T(2)$ (per $n \geq 6$ ci si riconduce a un caso per cui la condizione valga):

$$\begin{aligned} T(3) = 9 \cdot T(1) + 3 = 12 \leq c \cdot (3^2 - 3) \text{ per } c \geq 2 \\ T(4) = 9 \cdot T(1) + 4 = 13 \leq c \cdot (4^2 - 4) \text{ per } c \geq 13/12 \\ T(5) = 9 \cdot T(1) + 5 = 14 \leq c \cdot (5^2 - 5) \text{ per } c \geq 7/10 \end{aligned}$$

Affinché siano verificati tutti i casi base e il passo induttivo, dobbiamo avere $n_0 = 2$ e $c \geq 2$.

2.5.3 Il teorema fondamentale delle ricorrenze

Si tratta di un metodo per analizzare algoritmi basati sulla tecnica del *dividi et impera*, in cui:

- un problema di dimensione n viene diviso in a sottoproblemi di dimensione n/b;
- dividere in sottoproblemi e combinare le soluzioni richiede tempo $f(n)$.

La relazione di ricorrenza corrispondente a questo scenario è la seguente:

$$T(n) = a \cdot T\Big(\frac{n}{b}\Big) + f(n) \tag{2.1}$$

Per analizzare la relazione di ricorrenza, consideriamo l'albero della ricorsione ed assumiamo che n sia una potenza esatta di b e che la ricorsione si fermi quando $n = 1$.

Proprietà 2.5. I sottoproblemi al livello i dell'albero della ricorsione hanno dimensione n/b^i.

Proprietà 2.6. Il contributo di un nodo di livello i al tempo di esecuzione (escluso il tempo speso nelle chiamate ricorsive) è $f(n/b^i)$.

Proprietà 2.7. Il numero di livelli nell'albero della ricorsione è $\log_b n$.

Proprietà 2.8. Il numero di nodi al livello i dell'albero della ricorsione è a^i.

Usando le proprietà appena elencate, si può riscrivere la relazione di ricorrenza nella seguente forma:

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f\Big(\frac{n}{b^i}\Big) \tag{2.2}$$

La soluzione della (2.2) è data dal seguente teorema, noto come *teorema fondamentale delle ricorrenze*.

Teorema 2.2. *La relazione di ricorrenza*

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ a \cdot T(n/b) + f(n) & \text{se } n > 1 \end{cases}$$

ha soluzione:

1. $T(n) = \Theta(n^{\log_b a})$, *se* $f(n) = O(n^{\log_b a - \epsilon})$ *per* $\epsilon > 0$;
2. $T(n) = \Theta(n^{\log_b a} \cdot \log n)$, *se* $f(n) = \Theta(n^{\log_b a})$;
3. $T(n) = \Theta(f(n))$, *se* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *per* $\epsilon > 0$ *e* $a \cdot f(n/b) \leq c \cdot f(n)$ *per* $c < 1$ *ed* n *sufficientemente grande.*

Dimostrazione. Assumiamo per semplicità che n sia una potenza esatta di b.

Caso 1: riscriviamo il termine generico della sommatoria 2.2:

$$a^i \cdot f\Big(\frac{n}{b^i}\Big) = O\Big(a^i \cdot \Big(\frac{n}{b^i}\Big)^{\log_b a - \epsilon}\Big) = O\Big(n^{\log_b a - \epsilon} \cdot \Big(\frac{a \cdot b^\epsilon}{b^{\log_b a}}\Big)^i\Big) = O(n^{\log_b a - \epsilon} \cdot (b^\epsilon)^i)$$

Per limitare superiormente $T(n)$ si può scrivere:

$$T(n) = \sum_{i=0}^{\log_b n} O(n^{\log_b a - \epsilon} \cdot (b^\epsilon)^i) = O\left(n^{\log_b a - \epsilon} \cdot \sum_{i=0}^{\log_b n} (b^\epsilon)^i\right) = O\left(n^{\log_b a - \epsilon} \cdot \left(\frac{b^{\epsilon \cdot (\log_b n + 1)} - 1}{b^\epsilon - 1}\right)\right) =$$

$$O\left(n^{\log_b a - \epsilon} \cdot \left(\frac{b^\epsilon \cdot n^\epsilon - 1}{b^\epsilon - 1}\right)\right) = O(n^{\log_b a - \epsilon} \cdot n^\epsilon) = O(n^{\log_b a})$$

Analizzando l'equazione 2.2 e considerando solo i tempi di esecuzione relativi ai nodi sull'ultimo livello dell'albero di ricorsione, otteniamo:

$$T(n) \geq a^{\log_b n} = n^{\log_b a} \rightarrow T(n) = \Omega(n^{\log_b a})$$

Dalle due limitazioni segue che $T(n) = \Theta(n^{\log_b a})$.

Caso 2: anche in questo caso, riscriviamo il termine generico della sommatoria:

$$a^i \cdot f\Big(\frac{n}{b^i}\Big) = \Theta\Big(a^i \cdot \Big(\frac{n}{b^i}\Big)^{\log_b a}\Big) = \Theta\Big(n^{\log_b a} \cdot \Big(\frac{a}{b^{\log_b a}}\Big)^i\Big) = \Theta(n^{\log_b a})$$

Da cui segue:

$$T(n) = \sum_{i=0}^{\log_b n} \Theta(n^{\log_b a}) = \Theta(n^{\log_b a} \cdot \log_b n).$$

Caso 3: sotto l'assunzione $a \cdot f(n/b) \leq c \cdot f(n)$, risulta facile dimostrare che $a^i \cdot f(n/b^i) \leq c^i \cdot f(n)$; infatti:

$$a^i \cdot f\Big(\frac{n}{b^i}\Big) = a^{i-1} \cdot a \cdot f\Big(\frac{n/b^{i-1}}{b}\Big) \leq a^{i-1} \cdot c \cdot f\Big(\frac{n}{b^{i-1}}\Big)$$

Iterando il ragionamento si ottiene la disuguaglianza desiderata. Usando l'equazione 2.2, la serie geometrica con base $c < 1$ e la disuguaglianza appena dimostrata, si può scrivere:

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f\Big(\frac{n}{b^i}\Big) \leq f(n) \cdot \sum_{i=0}^{\infty} c^i = f(n) \cdot \frac{1}{1-c} = O(f(n))$$

Dalla relazione 2.2, si ricava immediatamente che $T(n) = \Omega(f(n))$, da cui $T(n) = \Theta(f(n))$. □

Esempio 2.4. Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ T(n/2) + O(1) & \text{altrimenti} \end{cases}$$

Si ha $f(n) = O(1) = \Theta(n^{\log_2 1})$; ci troviamo, dunque, nel caso 2 del teorema, e quindi risulta $T(n) = \Theta(\log n)$.

Esempio 2.5. Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1, 2 \\ 9 \cdot T(n/3) + n & \text{altrimenti} \end{cases}$$

Si ha $f(n) = n = O(n^{\log_3 9 - \epsilon})$; dal caso 1 del teorema fondamentale, risulta che $T(n) = \Theta(n^2)$.

Esempio 2.6. Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ T(n/2) + n + \log_n + 1 \end{cases}$$

Si ha $f(n) = n + \log n + 1 = \Theta(n) = \Omega(n^{\log_2 1 - \epsilon})$, che può ricadere nel caso 3 del teorema fondamentale; occorre infatti trovare $c > 1$ tale che, per n sufficientemente grande, valga:

$$\begin{aligned} & a \cdot f(n/b) \leq c \cdot f(n) \\ & \to 1 \cdot \left(\frac{n}{2} + \log \frac{n}{2} + 1 \right) \leq c \cdot (n + \log n + 1) \\ & \to c \geq \frac{\frac{n}{2} + \log \frac{n}{2} + 1}{n + \log n + 1} \left(= \frac{1}{2} < 1 \text{ per } n \to +\infty \right) \end{aligned}$$

Esisterà n_0 tale che, per $n \geq n_0$, si ha

$$\frac{\frac{n}{2} + \log \frac{n}{2} + 1}{n + \log n + 1} \leq \frac{3}{4}$$

È sufficiente prendere $c = 3/4$ affinché la disuguaglianza del caso 3 del teorema fondamentale sia verificata per n sufficientemente grande; segue che $T(n) = \Theta(n)$.

Esempio 2.7. Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1, 2 \\ 3 \cdot T(n/3) + n \cdot \log n & \text{altrimenti} \end{cases}$$

Nessuno dei casi del teorema principale può essere applicato; si potrebbe pensare di utilizzare il terzo caso, ma questo non è possibile poiché $n \cdot \log n$ è solo logaritmicamente, e non polinomialmente, più grande di $n^{\log_3 3} = n$.

2.5.4 Analisi dell'albero della ricorsione

La tecnica consiste nell'analizzare l'albero delle chiamate ricorsive, indicando le dimensioni dei problemi di ogni chiamata ricorsiva, ed analizzando la dimensione totale dei problemi ad ogni livello dell'albero.

Esempio 2.8. Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n = 1, 2 \\ 9 \cdot T(n/3) + n^2 \cdot \log n & \text{altrimenti} \end{cases}$$

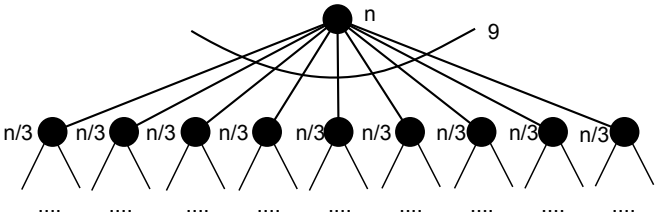


Figura 2.1: Parte iniziale dell'albero di ricorsione

Calcoliamo il costo di ciascun livello dell'albero di ricorsione:

Liv. 0: problema di dimensione n e costo $n^2 \cdot \log n \ \rightarrow$ totale $= n^2 \cdot \log n$;

Liv. 1: problema di dimensione $\frac{n}{3}$ e costo $\left(\frac{n}{3}\right)^2 \cdot \log \frac{n}{3} \rightarrow$ totale $= 9 \cdot \left(\frac{n}{3}\right)^2 \cdot \log \frac{n}{3}$;

Liv. 2: problema di dimensione $\frac{n}{3^2}$ e costo $\left(\frac{n}{3^2}\right)^2 \cdot \log \frac{n}{3^2} \rightarrow$ totale $= 9^2 \cdot \left(\frac{n}{3^2}\right)^2 \cdot \log \frac{n}{3^2}$;

Liv. i: problema di dimensione $\frac{n}{3^i}$ e costo $\left(\frac{n}{3^i}\right)^2 \cdot \log \frac{n}{3^i} \rightarrow$ totale $= 9^i \cdot \left(\frac{n}{3^i}\right)^2 \cdot \log \frac{n}{3^i} = n^2 \cdot \log n - i \cdot n^2 \cdot \log 3)$;

Foglie: problemi di dimensione $1 \rightarrow \frac{n}{3^k} = 1 \rightarrow k = \log_3 n$.

$$
\begin{aligned}
T(n) &= \underbrace{\sum_{i=0}^{\log_3 n - 1} (n^2 \cdot \log n - n^2 \cdot i \cdot \log 3)}_{\text{contributo nodi interni}} + \underbrace{9^{\log_3 n}}_{\text{contributo foglie}} \\
&= n^2 \cdot \log n \cdot \sum_{i=0}^{\log_3 n - 1} 1 - n^2 \cdot \log 3 \cdot \sum_{i=0}^{\log_3 n - 1} i + n^2 \\
&= n^2 \cdot \log n \cdot \log_3 n - n^2 \cdot \log 3 \cdot \frac{(\log_3 n - 1) \cdot (\log_3 n)}{2} + n^2 \\
&= n^2 \cdot \log n \cdot \log_3 n - n^2 \cdot \log 3 \cdot \frac{\log_3^2 n}{2} + n^2 \cdot \log 3 \cdot \frac{\log_3 n}{2} + n^2
\end{aligned}
$$

Risulta quindi $T(n) = \Theta(n^2 \cdot \log^2 n)$.

2.5.5 Cambiamenti di variabile

Si tratta di una tecnica che viene utilizzata quando la relazione di ricorrenza presenta delle radici; per maggiori informazioni, vedere l'esempio.

Esempio 2.9. Consideriamo la seguente relazione di ricorrenza:

$$
T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ T(\sqrt{n}) + O(1) & \text{altrimenti} \end{cases}
$$

Eseguiamo la sostituzione $n = 2^x$ (ossia $x = \log n$), da cui $\sqrt{n} = 2^{x/2}$ e $T(2^x) = T(2^{x/2}) + O(1)$; poniamo inoltre $T(2^x) = R(x)$, da cui otteniamo la relazione di ricorrenza $R(x) = R(x/2) + O(1)$, risolvibile con il teorema principale ($R(x) = O(\log x)$). Combinando le uguaglianze $T(2^x) = O(\log x)$ e $n = 2^x$, si ha $T(n) = O(\log \log n)$.

Capitolo 3

Correttezza degli algoritmi

3.1 Algoritmi iterativi

Per dimostrare la correttezza degli algoritmi iterativi si utilizza l'*invariante di ciclo*, ossia una proposizione (riguardante i contenuti delle variabili della procedura o programma) che rispetta le seguenti proprietà:

Inizializzazione: la proposizione è vera immediatamente prima di entrare nel ciclo.

Mantenimento: se la proposizione è vera prima di eseguire un'iterazione, lo è anche al termine dell'iterazione.

Terminazione: al termine del ciclo, la proposizione permette di ricavare la proprietà che permette di dimostrare la correttezza dell'algoritmo.

Esempio 3.1. Consideriamo l'algoritmo `fibonacciIterativoModificato`, del quale vogliamo dimostrare il fatto che calcoli l'n-esimo numero di Fibonacci; l'invariante di ciclo è il seguente:

$$\text{Ad ogni iterazione, } b = F_{i-1} \text{ e } a = F_{i-2}$$

Inizializzazione: poiché $i = 3$, dobbiamo verificare che $b = F_2$ e $a = F_1$; la dimostrazione è immediata, poiché $b = 1 = F_2$ e $a = 1 = F_1$.

Mantenimento: assumiamo l'invariante verificato per una generica i-esima iterazione e dimostriamo che esso vale anche al termine di tale iterazione; ricordiamo inoltre, nonostante non sia esplicitamente indicato, che al termine dell'iterazione viene incrementato l'indice i. Grazie alla nostra assunzione, abbiamo $b = F_{i-1}$ e $a = F_{i-2}$: alla variabile c viene assegnato il valore della somma $a + b = F_{i-2} + F_{i-1} = F_i$, alla variabile a viene assegnato il valore di $b = F_{i-1}$, alla variabile b il valore di $c = F_i$ ed i viene incrementata ($i = i+1$); è immediato dimostrare che continua a valere l'invariante, ossia che $b = F_{(i+1)-1}$ e $a = F_{(i+1)-2}$.

Terminazione: all'uscita del ciclo, si ha $i = n+1$, dunque $b = F_{(n+1)-1} = F_n$, che è il valore restituito, come volevasi dimostrare.

3.2 Algoritmi ricorsivi

La dimostrazione viene svolta procedendo per induzione; occorre formalizzare una proprietà utile per dimostrare la correttezza dell'algoritmo e provare che:

- valga per i *casi base*;
- assumendo che valga per problemi di dimensione inferiore, ossia per le chiamate ricorsive eseguite, provare che vale anche per il problema iniziale (*passo induttivo*).

Esempio 3.2. Consideriamo l'algoritmo `fibonacciRicorsivo` e formalizziamo la seguente proprietà:

L'output della chiamata di funzione `fibonacciRicorsivo`(n) è l'n-esimo numero di Fibonacci.

Casi base: per $n = 1$ e $n = 2$, l'algoritmo restituisce $1 = F_1 = F_2$.

Ipotesi induttiva: supponiamo che, per $k < n$, `fibonacciRicorsivo`(k) restituisca F_k.

Passo induttivo: l'algoritmo restituisce `fibonacciRicorsivo`$(n-1)$ + `fibonacciRicorsivo`$(n-2)$ (sia $n > 2$); per ipotesi, essi restituiscono, rispettivamente, F_{n-1} e F_{n-2}, la cui somma è F_n, come volevasi dimostrare.

Capitolo 4

Pile e code

4.1 Pile

La *pila* è una struttura dati, realizzabile sia con strutture indicizzate, sia collegate, che può essere descritta dal seguente schema generale:

Dati: una sequenza S di n elementi.

Operazioni:

`isEmpty()` $\rightarrow$ *booleano*
Restituisce *true* se S è vuota, *false* altrimenti.

`push(`*elem e*`)` $\rightarrow$ *void*
Aggiunge e come ultimo elemento di S.

`pop()` $\rightarrow$ *elem*
Toglie da S l'ultimo elemento e lo restituisce.

`top()` $\rightarrow$ *elem*
Restituisce l'ultimo elemento di S, senza rimuoverlo.

4.2 Code

La *coda*, come la pila, è una struttura dati realizzabile sia mediante strutture indicizzate, sia con strutture collegate; la realizzazione di una coda segue il seguente schema generale:

Dati: una sequenza S di n elementi.

Operazioni:

`isEmpty()` $\rightarrow$ *booleano*
Restituisce *true* se S è vuota, *false* altrimenti.

`enqueue(`*elem e*`)` $\rightarrow$ *void*
Aggiunge e come ultimo elemento di S.

`dequeue()` $\rightarrow$ *elem*
Toglie da S il primo elemento e lo restituisce.

`first()` $\rightarrow$ *elem*
Restituisce il primo elemento di S, senza rimuoverlo.

Capitolo 5

Alberi

5.1 Introduzione

Definizione 5.1. Un *albero* è una coppia $T = (N, A)$ costituita da un insieme N di *nodi* e da un insieme $A \subseteq N \times N$ di coppie di nodi, dette *archi*.

In un albero, ogni nodo v (esclusa la *radice*) ha un solo *padre* tale che $(u, v) \in A$; ogni nodo, inoltre, può avere un certo numero di figli w tali che $(v, w) \in A$, ed il loro numero è detto *grado* del nodo. Un nodo senza figli è chiamato *foglia* e tutti i nodi che non sono né foglia né radice sono detti *nodi interni*.
La *profondità* di un nodo è definita come segue:

- la radice ha profondità zero;
- se un nodo ha profondità k, i suoi figli avranno profondità $k + 1$.

I nodi che hanno lo stesso padre sono detti *fratelli*, e dunque avrannò la stessa profondità. L'*altezza* di un albero è definita come la massima profondità tra quelle delle varie foglie.
Un albero *d-ario* è un albero in cui tutti i nodi tranne le foglie hanno grado d; se tutte hanno medesima profondità, si dice che è *completo*.
Lo schema generale delle operazioni eseguibili su un albero è il seguente:

Dati: un insieme di nodi e un insieme di archi

Operazioni:

`numNodi()` $\rightarrow$ *intero*
Restituisce il numero di nodi presenti nell'albero.

`grado(`*nodo v*`)` $\rightarrow$ *intero*
Restituisce il numero di figli del nodo v.

`padre(`*nodo v*`)` $\rightarrow$ *nodo*
Restituisce il padre del nodo v nell'albero, *null* se v è la radice.

`figli(`*nodo v*`)` $\rightarrow \langle nodo, nodo, ..., nodo \rangle$
Restituisce i figli del nodo v.

`aggiungiNodo(`*nodo u*`)` $\rightarrow$ *nodo*
Inserisce un nuovo nodo $v$come figlio di u e lo restituisce. Se v è il primo nodo ad essere inserito nell'albero, diventa la radice.

`aggiungiSottoalbero(`*albero a, nodo u*`)` $\rightarrow$ *albero*
Inserisce nell'albero il sottoalbero a in modo che la sua radice diventi figlia di u.

`rimuoviSottoalbero(`*nodo v*`)` $\rightarrow$ *albero*
Stacca e restituisce l'intero sottoalbero radicato in v.

5.2 Rappresentazioni

Le modalità di rappresentazione di un albero possono essere essenzialmente suddivise in due categorie: le *rappresentazioni indicizzate* e le *rappresentazioni collegate*.
Le prime, nonostante risultino essere di facile realizzazione, rendono difficoltoso l'inserimento e la cancellazione di nodi nell'albero, mentre le seconde risultano essere decisamente più flessibili, nonostante siano leggermente più complesse da implementare.

5.2.1 Rappresentazioni indicizzate

Vettore padri

La più semplice rappresentazione possibile per un albero $T = (N, A)$ con n nodi è quella basata sul *vettore padri*: è un array di dimensione n le cui celle contengono coppie (*info*, *parent*), dove *info* è il contenuto informativo del nodo e *parent* il riferimento al padre (o *null* se si tratta della radice). Con questa implementazione, da ogni nodo è possibile risalire al padre in tempo $O(1)$, ma la ricerca dei figli richiede tempo $O(n)$.

Vettore posizionale

Consideriamo un albero d-ario completo con n nodi, dove $d \geq 2$; un *vettore posizionale* è un array P di dimensione n tale che $P[v]$ contiene l'informazione associata al nodo v e i figli sono memorizzati nelle posizioni $P[d \cdot v + i]$, con $0 \leq i \leq d - 1$. Da ciascun nodo è possibile risalire in tempo costante sia al proprio padre (indice $\lfloor v/d \rfloor$ se v non è la radice), sia a uno qualsiasi dei propri figli.

5.2.2 Rappresentazioni collegate

Puntatori ai figli

Se ogni nodo dell'albero ha al più grado d, è possibile mantenere in ogni nodo un puntatore a ciascuno dei possibili figli (se il figlio non è presente, si imposta a *null* il riferimento).

Lista figli

Se il numero massimo di figli non è noto a priori, si può mantenere per ogni nodo una lista di puntatori ai figli.

Primo figlio - fratello successivo

Variante della soluzione precedente, prevede di mantenere per ogni nodo un puntatore al primo figlio e uno al fratello successivo (*null* se non è presente, rispettivamente, il figlio o fratello); per scandire tutti i figli di un nodo, è sufficiente visitare il primo figlio e poi tutti i suoi fratelli.

5.3 Visite

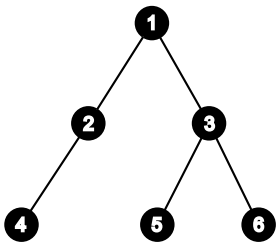


Figura 5.1: Esempio di albero

5.3.1 Visita in profondità

In una visita in profondità, si prosegue la visita dall'ultimo nodo lasciato in sospeso: può essere realizzata mediante l'utilizzo di *pile* o, in maniera più semplice, usando la ricorsione.

algoritmo `visitaSimmetrica` (*nodo r*) → *void*

```
1 if(r != null) then
2   visitaSimmetrica(figlio sinistro di r)
3   visita il nodo r
4   visitaSimmetrica(figlio destro di r)
```

Tre varianti classiche della visita in profondità sono le seguenti:

Visita in preordine: si visita prima la radice, poi vengono eseguite le chiamate ricorsive sul figlio sinistro e destro (Figura 5.1 → 1, 2, 4, 3, 5, 6)

Visita simmetrica: si effettua prima la chiamata sul figlio sinistro, poi si visita la radice e infine si esegue la chiamata ricorsiva sul figlio destro (Figura 5.1 → 4, 2, 1, 5, 3, 6)

Visita in postordine: si effettuano prima le chiamate ricorsive sul figlio sinistro e destro, poi viene visitata la radice (Figura 5.1 → 4, 2, 5, 6, 3, 1)

5.3.2 Visita in ampiezza

La visita in ampiezza è realizzata tramite l'uso di *code* e la sua caratteristica principale è il fatto che i nodi vengono visitati per livelli: l'ordine di visita dell'albero rappresentato in Figura 5.1 è 1, 2, 3, 4, 5, 6.

algoritmo `visitaAmpiezza` (*nodo r*) → *void*

```
1 Coda c
2 c.enqueue(r)
3 while(!c.isEmpty()) do
4   u = c.dequeue()
5   if(u != null)
6     visita il nodo u
7     c.enqueue(figlio sinistro di u)
8     c.enqueue(figlio destro di u)
```

5.4 Alberi binari di ricerca

Definizione 5.2. Un *albero binario di ricerca* è un albero binario che soddisfa le seguenti proprietà:

- ogni nodo v contiene un elemento *elem(v)* cui è associata una chiave *chiave(v)* presa da un dominio totalmente ordinato;
- le chiavi nel sottoalbero sinistro di v sono minori o uguali a *chiave(v)*;
- le chiavi nel sottoalbero destro di v sono maggiori o uguali a *chiave(v)*.

Un albero binario di ricerca è descritto dal seguente schema generale:

Dati: un albero binario di ricerca di altezza h e n nodi, ciascuno contenente coppie (*elem*, *chiave*).

Operazioni:

`search(`*chiave k*`)` → *elem*
Partendo dalla radice, ricerca un elemento con chiave k, usando la proprietà di ricerca per decidere quale sottoalbero esplorare (*complessità* $O(h)$).

`insert(`*elem e, chiave k*`)` → *void*
Crea un nuovo nodo v contenente la coppia (e, k) e lo aggiunge all'albero in posizione opportuna, in modo da mantenere la proprietà di ricerca (*complessità* $O(h)$).

`delete(`*elem e*`)` $\rightarrow$ *void*
Se il nodo v contenente l'elemento e ha al più un figlio, elimina v collegando il figlio all'eventuale padre, altrimenti scambia il nodo v con il suo predecessore ed elimina il predecessore (*complessità* $O(h)$).

5.5 Alberi AVL

Nel caso peggiore, l'altezza di un albero binario di ricerca può essere proporzionale al numero n di nodi, mentre, se fosse bilanciato, avrebbe altezza logaritmica, migliorando l'efficienza delle operazioni; per risolvere il problema, vengono usati gli *alberi AVL*.

Definizione 5.3. Un albero è *bilanciato in altezza* se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono al più di un'unità.

Definizione 5.4. Il *fattore di bilanciamento* $\beta(v)$ di un nodo v è la differenza tra l'altezza del sottoalbero sinistro e quella del sottoalbero destro di v:

$$\beta(v) = altezza(sin(v)) - altezza(des(v)) \tag{5.1}$$

Un albero AVL è un albero bilanciato in altezza e, oltre all'elemento e alla chiave, ciascun nodo mantiene l'informazione sul fattore di bilanciamento. Per mantenere l'albero bilanciato a seguito di cancellazioni ed inserimenti, occorre operare, qualora risultasse necessario, delle *rotazioni*.

5.5.1 Ribilanciamento tramite rotazioni

Le operazioni di rotazione vengono effettuate su nodi sbilanciati, ossia nodi il cui fattore di bilanciamento, in valore assoluto, è maggiore o uguale a 2; si possono distinguere vari casi:

Sinistra - sinistra: si esegue quando un nodo ha coefficiente di bilanciamento +2 ed il figlio destro un coefficiente pari a 0 o a +1 (in Figura 5.2 il nodo sbilanciato è quello con chiave 3).

Destra - destra: si esegue quando un nodo ha coefficiente di bilanciamento -2 ed il figlio sinistro un coefficiente pari a 0 o -1 (in Figura 5.2 il nodo sbilanciato è quello con chiave 6).

Sinistra - destra: si esegue quando un nodo ha coefficiente di bilanciamento -2 e il figlio sinistro un coefficiente pari a +1 (in Figura 5.2 il nodo sbilanciato è quello con chiave 3).

Destra - sinistra: si esegue quando un nodo ha coefficiente di bilanciamento +2 e il figlio destro un coefficiente pari a -1 (in Figura 5.2 il nodo sbilanciato è quello con chiave 6).

Proprietà 5.1. Una rotazione SS, SD, DS o DD, applicata ad un nodo v con fattore di bilanciamento ± 2, fa decrescere di 1 l'altezza del sottoalbero radicato in v prima della rotazione.

Mentre l'operazione di ricerca si svolge esattamente come in un albero binario di ricerca, le operazioni di cancellazione e inserimento sono soggette a modifica:

Inserimento

- si crea un nuovo nodo e lo si inserisce nell'albero con lo stesso procedimento usato per i BST;
- si ricalcolano i fattori di bilanciamento che sono mutati in seguito all'inserimento (solo i fattori di bilanciamento dei nodi nel cammino tra la radice e il nuovo elemento possono mutare e possono essere facilmente calcolati risalendo nel cammino dalla foglia verso la radice);
- se nel cammino compare un nodo con fattore di bilanciamento pari a ± 2, occorre ribilanciare mediante rotazioni (si può dimostrare che è sufficiente una sola rotazione per bilanciare l'albero).

Cancellazione

- si cancella il nodo con il medesimo procedimento usato nei BST;
- si ricalcolano i fattori di bilanciamento mutati in seguito all'operazione (solo i nodi nel cammino tra la radice e il padre del nodo eliminato possono aver subito una modifica del fattore);
- per ogni nodo con fattore di bilanciamento pari a ± 2, procedendo dal basso verso l'alto, si opera una rotazione (può essere necessario eseguire più rotazioni, in questo caso).

5.5.2 Alberi di Fibonacci

Definizione 5.5. Tra tutti gli alberi di altezza h bilanciati in altezza, un *albero di Fibonacci* ha il minimo numero di nodi.

Un albero di Fibonacci di altezza h può essere costruito unendo, tramite l'aggiunta di una radice, un albero di Fibonacci di altezza $h-1$ e uno di altezza $h-2$; è facile verificare che ogni nodo interno di un albero di questo tipo ha fattore di bilanciamento pari a $+1$ (o -1, dipende dalla costruzione): studiamo ora il rapporto tra numero di nodi ed altezza di un albero di Fibonacci.

Lemma 5.1. *Sia T_h un albero di Fibonacci di altezza h e sia n_h il numero dei suoi nodi; risulta $h = \Theta(\log n_h)$*

Dimostrazione. Per dimostrare la tesi, proviamo prima che $n_h = F_{h+3} - 1$, per induzione su h.

Passo base: si ha $h = 0$ e $n_0 = 1 = 2 - 1 = F_3 - 1$.

Ipotesi induttiva: supponiamo $n_k = F_{k+3} - 1$ per $k < h$.

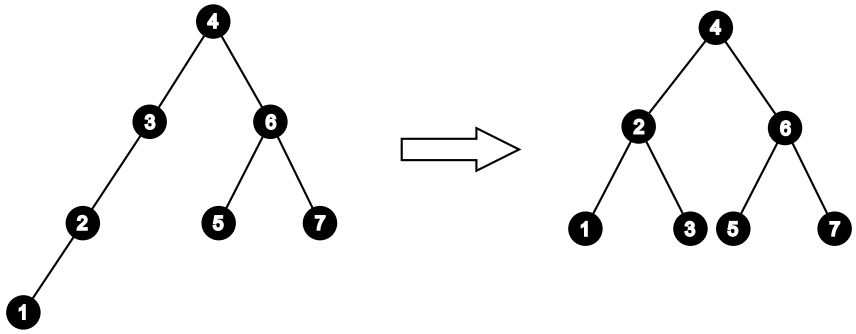
Passo induttivo: sia $h > 0$; si ha:

$$n_h = 1 + n_{h-1} + n_{h-2} = 1 + F_{h+2} - 1 + F_{h+1} - 1 = F_{h+3} - 1$$

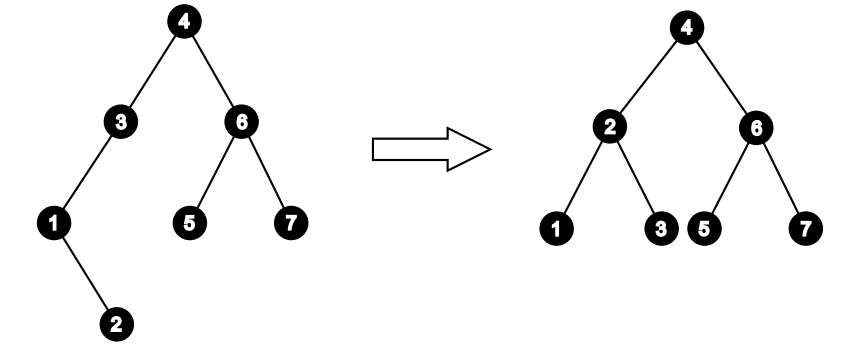
Poiché $F_h = \Theta(\phi^h)$, con $\phi \approx 1.618$, possiamo concludere che $h = \Theta(\log n_h)$. □

Corollario 5.1. *Un albero AVL con n nodi ha altezza $O(\log n)$.*

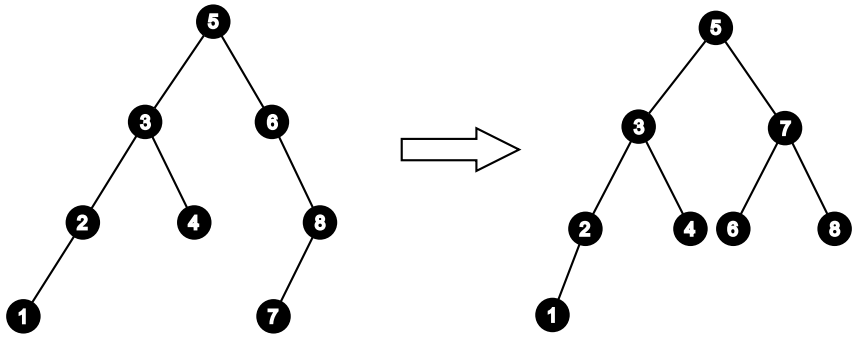
Dimostrazione. Sia h l'altezza dell'albero AVL; per dimostrare che $h = O(\log n)$ consideriamo l'albero di Fibonacci di altezza h, avente n_h nodi: per definizione di albero di Fibonacci si ha $n_h \leq n$ e, per il lemma appena dimostrato, si ottiene il risultato voluto. □



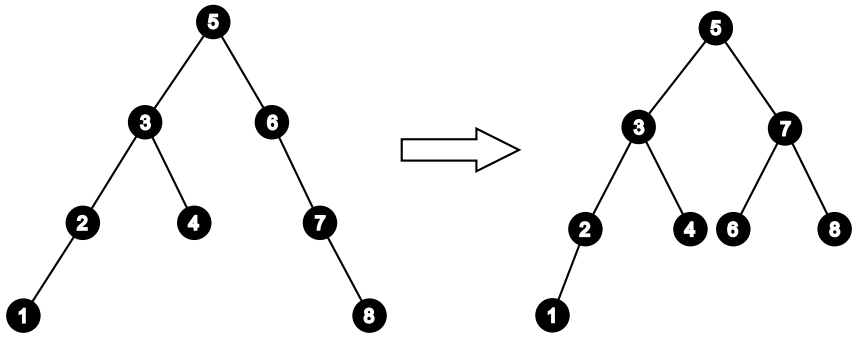
(a) Rotazione SS



(b) Rotazione SD



(c) Rotazione DS



(d) Rotazione DD

Figura 5.2: Esempi di rotazione

Capitolo 6

Heap e code di priorità

6.1 Heap

Definizione 6.1. Uno *heap* è un albero binario quasi completo che rispetta la *proprietà heap* definita come segue:

- nel caso di *min-heap*, il padre ha associata una chiave minore o uguale alle chiavi di tutti i suoi figli;
- nel caso di *max-heap*, il padre ha associata una chiave maggiore o uguale alle chiavi associati ai suoi figli.

Solitamente uno heap viene rappresentato utilizzando un vettore (a partire dalla posizione di indice 1) dove, per un generico nodo in posizione i, si ha:

- il padre in posizione $\lceil i/2 \rceil$;
- il figlio sinistro in posizione $2 \cdot i$;
- il figlio destro in posizione $2 \cdot i + 1$.

Lemma 6.1. *Uno heap con n nodi ha altezza $O(\log n)$.*

Dimostrazione. Sia h l'altezza dello heap con n nodi; poiché lo heap è completo almeno fino a profondità $h-1$, e un albero binario completo di profondità $h-1$ ha $\frac{2^h-1}{2-1} = 2^h - 1$ nodi, segue che $2^h - 1 < n$; con semplici passaggi algebrici, si giunge alla relazione $h < \log_2(n+1) = O(\log n)$. □

Nel seguito faremo riferimento solo a max-heap, ma quanto detto (con opportune modifiche dovute alla condizione heap diversa) può essere applicato anche a min-heap.

6.1.1 Costruzione

Per costruire un max-heap, si utilizza una funzione ausiliaria `heapifyDown`.

funzione ausiliaria `heapifyDown` (*vettore A, indice i*) → *void*

```
1 k = indiceArgomentoMassimo(i, figlioSx(i), figlioDx(i))
2 if(k != i)
3   swap(A[k], A[i])
4   heapifyDown(A, k)
```

Tale funzione verifica che il valore di $A[i]$ sia maggiore di quello dei figli, se presenti, e, in caso contrario, $A[i]$ migra verso il basso fino a che non vale la proprietà heap; la complessità è $O(h)$, dove h è l'altezza dell'albero. La funzione per la costruzione dello heap è la seguente:

funzione `buildHeap` (*vettore A*) → *void*

```
1 for i = length(A)/2} downto 1 do
2   heapifyDown(A, i)
```

Per dimostrare la correttezza della funzione, usiamo il seguente invariante di ciclo:

$$\forall k : i+1 \leq k \leq length(A), A[k] \text{ è radice di uno heap.}$$

Inizializzazione: si ha $i = length(A)/2$; $\forall k : i+1 \leq k \leq length(A), A[k]$ è una foglia e dunque, banalmente, radice di uno heap.

Mantenimento: assumiamo che la proprietà valga all'inizio di una generica iterazione e dimostriamo che vale anche alla fine; si ricorda che, ad ogni passo, i viene decrementata, nonostante non sia esplicitamente indicato. La chiamata `heapifyDown(`A`, `i`)` fa diventare $A[i]$ radice di uno heap, senza togliere questa proprietà agli altri: al termine dell'iterazione si ha che $A[(i-1)+1], A[(i-1)+2], ..., A[length(A)]$ sono radici di heap, ossia quanto volevamo dimostrare.

Terminazione: si ha $i = 0$ e che $A[1], A[2], ..., A[length(A)]$ sono tutti radici di heap, dunque A è uno heap.

Complessità

Per il calcolo della complessità, ci servono le seguenti formule:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \tag{6.1}$$

Dimostrazione.

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

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2} \tag{6.2}$$

Dimostrazione.

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

Per procedere, dobbiamo calcolare quante volte la funzione `buildHeap` richiama `heapify`; dividiamo i vertici in base all'altezza: per ogni altezza h ce ne sono al più $\frac{n}{2^{h+1}}$. Per nodi alla stessa altezza h, la complessità di `heapify` è $O(h)$; la complessità totale risulta essere:

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

Sostituendo $x = 1/2$ nell'equazione (6.2) otteniamo:

$$\sum_{h=0}^{\infty} \left\lceil \frac{h}{2^{h+1}} \right\rceil = \frac{1/2}{(1-1/2)^2} = 2$$

Segue che:

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

6.1.2 Realizzazione

Per realizzare le operazioni di inserimento e rimozione, risulta utile un'ulteriore funzione d'appoggio, simmetrica a `heapifyDown`.

funzione ausiliaria `heapifyUp` (*vettore A, indice i*) → *void*

```
1 if(i != 1 && A[i] < A[padre(i)]) then
2   swap(A[i], A[padre(i)])
3   heapifyUp(A, padre(i))
```

Inserimento: viene creato un nuovo nodo e inserito come foglia nella prima posizione libera del vettore usato per realizzare lo heap; a questo punto, per ripristinare la proprietà heap, viene richiamata la funzione `heapifyUp` sul nodo appena inserito. La complessità dell'operazione è $O(\log n)$.

Cancellazione: il nodo da rimuovere viene scambiato con l'ultimo elemento occupato del vettore; a questo punto, non sapendo se il nodo appena scambiato debba scendere o salire nell'albero, vengono richiamate su di esso entrambe le funzioni `heapifyUp` e `heapifyDown` per ripristinare la proprietà heap. La complessità dell'operazione, anche in questo caso, è $O(\log n)$.

6.2 Code di priorità

Le code di priorità possono essere realizzate usando degli heap: a seconda del fatto che sia usato un min-heap o un max-heap, si possono avere *code a min-priorità* e *code a max-priorità*. Una coda a min-priorità è descritta dal seguente schema generale:

Dati: un insieme S di n elementi di tipo *elem* a cui sono associate chiavi di tipo *chiave* prese da un universo totalmente ordinato.

Operazioni:

`findMin()` → *elem*
Restituisce l'elemento di S con la chiave minima.

`insert(`*elem e, chiave k*`)` → *void*
Aggiunge a S un nuovo elemento e con chiave K.

`delete(`*elem e*`)` → *void*
Cancella da S l'elemento e.

`deleteMin()` → *void*
Cancella da S l'elemento con chiave minima.

`increaseKey(`*elem e, chiave d*`)` → *void*
Incrementa della quantità d la chiave dell'elemento e in S.

`decreaseKey(`*elem e, chiave d*`)` $\rightarrow$ *void*
Decrementa della quantità d la chiave dell'elemento e in S.

Una coda a max-priorità, invece, al posto di funzioni per ricercare ed estrarre l'elemento con chiave minima, fornirà funzioni per ricercare ed estrarre quello con chiave massima.

Capitolo 7

Algoritmi di ordinamento

7.1 Selection sort

L'algoritmo di ordinamento per selezione opera nel modo seguente: supponiamo che i primi k elementi siano ordinati; l'algoritmo sceglie il minimo degli $n - k$ elementi non ancora ordinati e lo inserisce in posizione $k + 1$; partendo da $k = 0$ e iterando n volte il ragionamento, si ottiene la sequenza interamente ordinata.

algoritmo `selectionSort` (*vettore* A) $\rightarrow$ *void*

```
1 for k = 0 to length(A) - 2 do
2   m = k + 1
3   for j = m + 1 to length(A) do
4     if(A[j] < A[m]) then
5       m = j
6   swap(A[m], A[k + 1])
```

Complessità

L'estrazione del minimo (righe 2 - 4) richiede $n - k - 1$ confronti e, poiché il ciclo esterno viene eseguito $n - 1$ volte, si ha:

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

7.2 Insertion sort

L'algoritmo di ordinamento per inserzione opera in modo simile al precedente: supponiamo che i primi k elementi siano ordinati; l'algoritmo prende il $(k+1)$-esimo elemento e lo inserisce nella posizione corretta rispetto ai k elementi già ordinati; partendo da $k = 0$ ed iterando n volte il ragionamento, il vettore viene completamente ordinato.

algoritmo `insertionSort` (*vettore* A) $\rightarrow$ *void*

```
1 for k = 1 to length(A) - 1 do
2     value = A[k + 1]
3     j = k
4     while(j > 0 && A[j] > value) do
5         A[j + 1] = A[j]
6         j--
7     A[j + 1] = value
```

Complessità

Le righe 4 - 6 individuano la posizione in cui l'elemento va inserito, eseguendo al più k confronti; poiché il ciclo esterno viene eseguito $n-1$ volte, si ha:

$$T(n) = \sum_{k=1}^{n-1} k = O(n^2)$$

7.3 Bubble sort

L'algoritmo di ordinamento a bolle opera una serie di scansioni del vettore: in ogni scansione sono confrontate coppie di elementi adiacenti e viene effettuato uno scambio se i due elementi non rispettano l'ordinamento; se durante una scansione non viene eseguito alcuno scambio, allora il vettore è ordinato.

algoritmo `bubbleSort` (*vettore* A) $\rightarrow$ *void*

```
1 for i = 1 to length(A) - 1 do
2   for j = 2 to (n - i + 1) do
3     if(A[j - 1] > A[j])
4       swap(A[j - 1], A[j])
5     if(non ci sono stati scambi)
6       break
```

Lemma 7.1. *Dopo l'i-esima scansione, gli elementi $A[n-i+1], ..., A[n]$ sono correttamente ordinati e occupano la loro posizione definitiva, ovvero $\forall h, k$ con $1 \leq h \leq k \wedge n-i+1 \leq k \leq n$, risulta $A[h] \leq A[k]$.*

Dimostrazione. Procediamo per induzione sul numero i dell'iterazione.

Caso base: si ha $i = 1$; dopo la prima scansione, l'elemento massimo di A ha raggiunto l'n-esima posizione, quindi il passo base è facilmente verificato.

Ipotesi induttiva: supponiamo che l'enunciato valga per le prime $i-1$ scansioni.

Passo induttivo: verifichiamo la validità della proprietà dopo l'i-esima scansione; al termine di tale scansione, il massimo degli elementi $A[1], ..., A[n-i+1]$ avrà raggiunto la posizione $n-i+1$ e quindi, per ogni $h < n-i+1$, risulta $A[h] \leq A[n-i+1]$; combinando questa osservazione con l'ipotesi induttiva, si può verificare facilmente la validità dell'enunciato.

Complessità

Per il lemma appena dimostrato, dopo al più $n-1$ cicli il vettore è ordinato e, nel caso peggiore, all'i-esima scansione vengono eseguiti $n-i$ confronti; si ottiene quindi:

$$T(n) = \sum_{i=1}^{n-1} (n-i) = \sum_{j=1}^{n-1} j = O(n^2)$$

7.4 Heap sort

L'algoritmo di ordinamento heap sort, come dice il nome, si basa sull'utilizzo di una struttura dati di tipo heap.

algoritmo `heapSort` (*vettore* A) $\rightarrow$ *void*

```
1 buildHeap(A)
2 for i = length(A) - 1 downto 1 do
3   swap(A[i], A[1])
4   heapifyDown(A[1..(i - 1)], 1)
```

Complessità

La funzione `buildHeap` ha complessità $O(n)$, mentre `heapifyDown`, che viene richiamata n volte, ha complessità $O(\log n)$; in totale si ha:

$$T(n) = O(n) + n \cdot O(\log n) = O(n \cdot \log n)$$

7.5 Merge sort

L'algoritmo di ordinamento per fusione si basa sulla tecnica del *divide et impera*:

Divide: si divide il vettore in due parti di uguale dimensione; se ha un numero dispari di elementi, una delle due parti conterrà un elemento in più.

Impera: supponendo di avere le due sottosequenze ordinate, si fondono in una sola sequenza ordinata.

algoritmo `mergeSort` (*vettore A, indice i, indice f*) → *void*

```
1  if(i < f)
2    m = (i + f) / 2
3    mergeSort(A, i, m)
4    mergeSort(A, m + 1, f)
5    merge(A, i, m, f)

6  merge(vettore A, indice i1, indice f1, indice f2)
7  X: vettore[f2 - i1 + 1]
8  i = 1
9  i2 = f1 + 1
10 while(i1 < f1 && i2 < f2) do
11   if(A[i1] <= A[i2]) then
12     X[i] = A[i1]
13     i++, i1++
14   else
15     X[i] = A[i2]
16     i++, i2++
17 if(i1 < f1)
18   copia A[i1..f1] alla fine di X
19 else
20   copia A[i2..f2] alla fine di X
21 copia X in A[i1..f2]
```

Complessità

La complessità della funzione ausiliaria `merge` è $O(n)$; la relazione di ricorrenza è la seguente:

$$T(n) = 2 \cdot T(n/2) + O(n)$$

Applicando il secondo caso del teorema fondamentale, si ottiene:

$$T(n) = \Theta(n \cdot \log n)$$

Per quanto riguarda lo spazio di memoria, `mergeSort` non opera in loco e pertanto ha un'occupazione di memoria pari a $2 \cdot n$.

7.6 Quicksort

Anche questo tipo di algoritmo segue un approccio di tipo *divide et impera*:

Divide: sceglie un elemento x della sequenza (il *perno*), partiziona la sequenza in due sottosequenze contenenti, rispettivamente, gli elementi minori o uguali a x e gli elementi maggiori di x, e ordina ricorsivamente le due sottosequenze.

Impera: concatena le sottosequenze ordinate.

algoritmo `quickSort` (*vettore A, indice i, indice f*) → *void*

```
1  if(i < f)
2    m = partition(A, i, f)
3    quickSort(A, i, m - 1)
4    quickSort(A, m + 1, f)

5  partition(vettore A, indice i, indice f) → indice
6  x = A[i]
7  inf = i
8  sup = f + 1
9  while(true) do
10   do inf++ while(A[inf] <= x)
11   do sup-- while(A[sup] > x)
12   if(inf < sup) then
13     swap(A[inf], A[sup])
14   else
15     break
16 swap(A[i], A[sup])
17 return sup
```

L'invariante della procedura `partition` è il seguente:

Proprietà 7.1. In ogni istante, gli elementi $A[i], ..., A[inf - 1]$ sono minori o uguali al perno, mentre gli elementi $A[sup + 1], ..., A[f]$ sono maggiori del perno.

Complessità

L'algoritmo ordina un vettore A di dimensione n suddividendolo in due sottovettori, A_1 e A_2, di dimensioni a e b tali che $a + b = n - 1$, ed ordinando ricorsivamente A_1 e A_2; poiché la partizione richiede $n - 1$ confronti, si può scrivere la seguente relazione di ricorrenza:

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

Nel caso peggiore, si ha che ad ogni passo viene scelto come perno l'elemento più piccolo (o più grande) della porzione di vettore considerato, riducendo la relazione di ricorrenza a:

$$T(n) = T(n - 1) + n - 1 = O(n^2)$$

Dal risultato sembra che `quickSort` sia un algoritmo inefficiente; l'idea per migliorarlo consiste nell'usare la *randomizzazione*. Supponiamo di scegliere il perno come il k-esimo elemento di A, dove k è *scelto a caso*: per calcolare il numero atteso di confronti, bisogna calcolare la somma dei tempi di esecuzione pesandoli in base alla probabilità di fare una certa scelta casuale; in questo caso, la scelta è il valore di k e la probabilità di scegliere il k-esimo elemento come perno, $\forall k, 1 \leq k \leq n$ è $1/n$. Otteniamo la seguente relazione di ricorrenza:

$$T(n) = \sum_{a=0}^{n-1} \frac{1}{n} \cdot (T(a) + T(n - a - 1) + n - 1) = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} \cdot T(a) \tag{7.1}$$

Dimostriamo, usando il metodo di sostituzione, che $T(n) = O(n \cdot \log n)$, ossia che esiste una costante $\alpha > 0$ tale per cui $C(n) \leq \alpha \cdot n \cdot \log n$ per ogni $n \geq n_0, n_0 > 0$.

Passo base: si ha $n = 1$; $T(1) = 0 = \alpha \cdot (1 \cdot \log 1)$ per ogni α.

Ipotesi induttiva: supponiamo che esista α tale per cui $T(i) \leq \alpha \cdot i \cdot \log i$ per $i < n$.

Passo induttivo: usando l'ipotesi induttiva nella relazione 7.1 e la definizione di somme integrali, si ottiene:

$$T(n) = n-1+\sum_{i=0}^{n-1} \frac{2}{n} \cdot T(i) \leq n-1+\sum_{i=0}^{n-1} \frac{2}{n} \cdot \alpha \cdot i \cdot \log i = n-1+\frac{2 \cdot \alpha}{n} \cdot \sum_{i=2}^{n-1} i \cdot \log i \leq n-1+\frac{2 \cdot \alpha}{n} \cdot \int_2^n x \cdot \log x dx$$

Procediamo usando il metodo di integrazione per parti:

$$T(n) \leq n-1+\frac{2\cdot\alpha}{n}\cdot\left(n^2\cdot\frac{\log n}{2}-\frac{n^2}{4}+2\cdot\ln 2+1\right)=n-1+\alpha\cdot n\cdot\log n-\alpha\cdot\frac{n}{2}-O(1)\leq\alpha\cdot n\cdot\log n$$

quando $n-1<\alpha\cdot(n/2)$, ossia per $\alpha\geq 2$.

Possiamo dunque concludere che l'algoritmo `quickSort`, su un input di dimensione n ha complessità $O(n^2)$ nel caso peggiore, ma il numero atteso di confronti è $O(n\cdot\log n)$.

7.7 Counting sort

Si tratta di un algoritmo per ordinare vettori di interi di n elementi compresi nell'intervallo [1, k].

algoritmo `countingSort` (*vettore A, intero k*) → *void*

```
1  X: vettore[k]
2  for i = 1 to k do
3    X[i] = 0
4  for i = 1 to n do
5    X[A[i]]++
6  j = 1
7  for i = 1 to n do
8    while(X[i] > 0) do
9      A[j] = i
10     j++, X[i]--
```

Complessità

Il primo ciclo ha complessità $O(k)$, il secondo $O(n)$ e il terzo, nonostante sia costituito da due cicli innestati, è anch'esso $O(n)$; per rendersene conto, basta osservare che il tempo necessario per eseguire gli n incrementi dev'essere uguale a quello necessario per gli n decrementi, e gli incrementi sono eseguiti dal secondo ciclo. In totale la complessità risulta essere:

$$T(n)=O(n+k)$$

L'algoritmo `countingSort` è assai efficiente se $k=O(n)$, mentre è preferibile usarne altri all'aumentare dell'ordine di grandezza di k; si osservi, inoltre, che la memoria occupata è anch'essa $O(n+k)$.

7.8 Radix sort

Si tratta di un algoritmo per ordinare interi in tempo lineare, quando l'intervallo dei numeri che possono essere presenti nel vettore è troppo grande per pensare di utilizzare il `countingSort`; consiste nell'ordinare, ad ogni ciclo, in base all'i-esima cifra, partendo da quella meno significativa fino a quella più significativa.

algoritmo `radixSort` (*vettore A*) → *void*

```
1  t = 0
2  while(esiste un numero con la t-esima cifra diversa da 0)
3    bucketSort(A, 10, t)
4    t++

5  bucketSort(vettore A, intero b, intero t)
6  Y: vettore[b]
7  for i = 1 to b do
8    Y[i] = lista vuota
9  for i = 1 to n do
10   c = t-esima cifra di A[i] in base b
11   attacca A[i] alla lista Y[c + 1]
12 for i = 1 to b do
13   copia ordinatamente in A gli elementi di Y[i]
```

Complessità

Sia k il valore più grande presente nel vettore; l'algoritmo esegue $\log_{10} k$ chiamate di `bucketSort`, ciascuna delle quali ha costo $O(n)$, per un totale di $O(n \cdot \log k)$; è possibile aumentare il valore della base usata nel `bucketSort`, senza però aumentare significativamente il tempo di esecuzione, come afferma il seguente teorema.

Teorema 7.1. *Usando come base per il* `bucketSort` *un valore $b = \Theta(n)$, l'algoritmo* `radixSort` *ordina n interi in $[1, k]$ in tempo*

$$O\left(n \cdot \left(1 + \frac{\log k}{\log n}\right)\right)$$

Dimostrazione. Ci sono $\log_b k = O(\log_n k)$ passate di `bucketSort`, ciascuna delle quali dal costo $O(n)$: il tempo totale è $O(n \cdot \log_n k) = O(n \cdot \log k / \log n)$, per le regole del cambiamento di base dei logaritmi; per considerare il caso $k < n$, aggiungiamo $O(n)$ alla complessità, che è il tempo richiesto per leggere la sequenza. Il risultato della somma è quanto si voleva dimostrare. □

7.9 Limitazione inferiore per algoritmi basati sul confronto

Usiamo la seguente proprietà per stabilire la limitazione:

Proprietà 7.2. Il numero di confronti fatti dall'algoritmo A nel caso peggiore è pari all'altezza dell'albero di decisione, ovvero alla lunghezza del più lungo cammino dalla radice ad una foglia.

Lemma 7.2. *Un albero di decisione per l'ordinamento di n elemento contiene almeno $n!$ foglie.*

Dimostrazione. Ad ogni foglia dell'albero di decisione non ci sono più confronti da effettuare, dunque corrisponde ad una soluzione del problema dell'ordinamento; ogni soluzione corrisponde, a sua volta, ad una permutazione degli n elementi da ordinatre, quindi l'albero deve contenere un numero di foglie pari almeno al numero di permutazioni degli n elementi da ordinare, ossia $n!$ foglie. □

Lemma 7.3. *Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli e sia k il numero delle sue foglie: l'altezza è almeno $\log_2 k$.*

Dimostrazione. Definiamo $h(k)$ come l'altezza di un albero binario con k foglie (con 2 figli per ogni nodo interno) e dimostriamo il lemma per induzione su k.

Caso base: si ha $k = 1$; si tratta, dunque, di un albero con un solo nodo, quindi l'altezza è $0 \geq \log_2 1$.

Ipotesi induttiva: assumiamo che $h(i) \geq \log_2 i$ per $i < k$.

Passo induttivo: poiché uno dei sottoalberi ha almeno la metà delle foglie, si ha:

$$h(k) \geq 1 + h\left(\frac{k}{2}\right) \geq 1 + \log_2\left(\frac{k}{2}\right) = 1 + \log_2 k - \log_2 2 = \log_2 k$$

come volevasi dimostrare. □

Teorema 7.2. *Il numero di confronti necessari per ordinare n elementi nel caso peggiore è $\Omega(n \cdot \log n)$.*

Dimostrazione. Per il lemma 7.2, il numero di foglie di un albero di decisione è almeno $n!$ e, per la proprietà 7.2, il numero di confronti necessari per l'ordinamento è, nel caso peggiore, pari all'altezza dell'albero, ossia $\geq \log n!$ per il lemma 7.3. La disuguaglianza di De Moivre - Stirling afferma che, per n sufficientemente grande, si ha:

$$\sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \frac{1}{12 \cdot n - 1}\right)$$

Si ha dunque:

$$n! \approx \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n$$

da cui:

$$\log n! \approx n \cdot \log n - 1.4427 \cdot n - \frac{1}{2} \cdot \log n + 0.826$$

Da questa relazione segue direttamente la limitazione inferiore che si voleva dimostrare. □

Capitolo 8

Tabelle hash

Sono strutture dati realizzate mediante vettori che permettono, nel caso medio, di eseguire operazioni in tempo costante. Sia U l'universo delle chiavi associabili agli elementi che vogliamo memorizzare; nel caso in cui U non sia l'insieme di numeri naturali $[0, m-1]$, l'utilizzo di tavole ad accesso diretto (ossia tavole che usano direttamente la chiave come indice per reperire l'elemento nella tabella) risulta troppo costoso, a causa del *fattore di carico* troppo basso: per questo motivo vengono usate le *funzioni hash* per la trasformazione delle chiavi in indici.

Definizione 8.1. Il *fattore di carico* di una tavola è definito come il rapporto $\alpha = n/m$ tra il numero n di elementi in essa memorizzati e la sua dimensione m.

Definizione 8.2. Una *funzione hash* è una funzione $h : U \to \{0, ..., m-1\}$ che trasforma chiavi in indici di una tavola.

Definizione 8.3. Una funzione hash h è perfetta se è iniettiva, ovvero $\forall u, v \in U, u \neq v \Rightarrow h(u) \neq h(v)$.

Affinché una funzione hash sia perfetta, occorre che $|U| \leq m$, ossia ci dev'essere spazio per tanti elementi quante sono le chiavi possibili: questo comporta un enorme spreco di memoria se l'insieme delle chiavi è molto grande. Se una funzione hash non è perfetta, allora potrebbe verificarsi una *collisione*, ossia si possono avere più chiavi diverse con lo stesso valore associato: per risolvere il problema, occorre operare delle strategie di risoluzione delle collisioni, che hanno lo svantaggio di ridurre le prestazioni.

8.1 Definizione di funzioni hash

Definizione 8.4. Sia

$$Q(i) = \sum_{k:h(k)=i} P(k)$$

la probabilità che, scegliendo una chiave, questa finisca nella cella i; una funzione hash h gode della proprietà di *uniformità semplice* se, $\forall i \in \{0, ..., m-1\}$,

$$Q(i) = \frac{1}{m}$$

Per definire funzioni hash con buone caratteristiche di uniformità, con l'assunzione che ogni chiave abbia la stessa probabilità di essere scelta, si usano spesso le seguenti tecniche.

Metodo della divisione: il metodo calcola il resto della divisione della chiave k per m, dove m è la dimensione della tabella hash; sebbene nella maggior parte dei casi si hanno buoni risultati, in altri potrebbero verificarsi molte collisioni: la bontà del metodo dipende dalla scelta di m, che sarebbe preferibile fosse un numero primo vicino ad una potenza di due, e dal fatto che la funzione hash dovrebbe dipendere da tutti i bit della chiave.

Metodo del ripiegamento: consiste nel dividere la chiave k in l parti e definire la funzione hash come l'applicazione di una funzione f, con codominio $\{0, ..., m-1\}$, sulle parti di chiave ottenute con la divisione; ossia:

$$h(k) = f(k_1, k_2, ..., k_l)$$

8.2 Risoluzione delle collisioni

8.2.1 Liste di collisione

Questo metodo consiste nell'associare a ciascuna cella della tabella hash una lista di chiavi, detta *lista di collisione*, di lunghezza media pari al fattore di carico α; per questo motivo, assumendo che la funzione di hashing goda della proprietà di uniformità semplice, si ha che il tempo medio necessario per un'operazione di ricerca o eliminazione è $O(1+\alpha)$, mentre l'inserimento può essere realizzato in tempo $O(1)$.

8.2.2 Indirizzamento aperto

Nel caso in cui la posizione $h(k)$ in cui inserire una chiave k sia già occupata, il metodo prevede di posizionarla in un'altra cella vuota, anche se quest'ultima potrebbe spettare di diritto ad un'altra chiave. Le operazioni vengono realizzate come segue:

Inserimento:

- se $v[h(k)]$ è vuota, inserisci la coppia (el, k) in tale posizione;
- altrimenti, a partire da $h(k)$, ispeziona le celle della tabella secondo una sequenza opportuna di indici $c(k,0)$, $c(k,1)$, ..., $c(k,m-1)$ e inserisci nella prima cella vuota; la sequenza, chiaramente, deve contenere tutti gli indici $\{0,...,m-1\}$.

Ricerca:

- se, durante la scansione delle celle, ne viene trovata una con la chiave cercata, restituisci l'elemento trovato;
- altrimenti, se si arriva a una cella vuota o si è scandita l'intera tabella senza successo, restituisci *null*.

Cancellazione: affinché la ricerca con il metodo appena descritto funzioni, occorre adottare una strategia particolare per la cancellazione, ossia utilizzare un valore speciale *canc* nel campo *el* dell'elemento che si vuole rimuovere: in particolare, l'inserimento tratterà tale cella come vuota e si fermerà su di essa, mentre la ricerca la oltrepasserà.

Le prestazioni delle operazioni implementate dipendono dalla particolare funzione $c(k,i)$ usata, ossia dal tipo di scansione scelto:

Scansione lineare:

$$c(k,i) = (h(k)+i) \bmod m \text{ con } 0 \leq i < m$$

Dopo un certo numero di inserimenti, tendono a formarsi degli agglomerati sempre più lunghi di celle piene, che comportano un decadimento delle prestazioni; si parla del problema di di *agglomerazione primaria*.

Scansione quadratica:

$$c(k,i) = \left\lfloor h(k) + c_1 \cdot i + c_2 \cdot i^2 \right\rfloor \bmod m, \text{ con } 0 \leq i < m \text{ e } c_1 \text{ e } c_2 \text{ opportuni}$$

Nonostante la scansione quadratica distribuisca le chiavi in modo da evitare l'agglomerazione primaria, ogni coppia di chiavi k_1 e k_2 con $h(k_1) = h(k_2)$ continua a generare la stessa sequenza di scansione; questo dà luogo all'*agglomerazione secondaria*.

Hashing doppio:

$$c(k,i) = \lfloor h_1(k) + i \cdot h_2(k) \rfloor \bmod m$$

con h_1 e h_2 funzioni hash distinte e m e $h_2(k)$ primi tra loro. Si tratta di un metodo che permette di eliminare virtualmente il fenomeno dell'agglomerazione, facendo dipendere dalla chiave anche il passo dell'incremento dell'indice usando una seconda funzione hash.

Complessità

Nell'ipotesi che le chiavi associate agli elementi di una tavola hash siano prese dall'universo delle chiavi con probabilità uniforme, il numero medio di passi richiesto da un'operazione di ricerca (contando anche le celle marcate come *canc*) è descritto nella seguente tabella:

Esito ricerca	*Scansione lineare*	*Scansione quadratica / Hashing doppio*
Chiave trovata	$\frac{1}{2} + \frac{1}{2 \cdot (1-\alpha)}$	$-\frac{1}{\alpha} \cdot \ln(1-\alpha)$
Chiave non trovata	$\frac{1}{2} + \frac{1}{2 \cdot (1-\alpha)^2}$	$\frac{1}{1-\alpha}$

Capitolo 9

Tecniche algoritmiche

9.1 Divide et impera

Si tratta di una tecnica già incontrata in precedenza, utilizzata, ad esempio, dagli algoritmi di ordinamento `mergeSort` e `quickSort`; il principio su cui si basa questa tecnica consiste nel dividere i dati in ingresso in un certo numero di sottoinsiemi (*divide*), risolvere ricorsivamente il problema sui sottoinsiemi e ricombinare le sottosoluzioni così ottenute per ottenere la soluzione del problema originario (*impera*).

9.2 Programmazione dinamica

Si tratta di una tecnica *bottom-up* introdotta, in maniera informale, con l'algoritmo `fibonacciIterativo`, che si basa sull'utilizzo di una tabella per memorizzare le soluzioni dei sottoproblemi incontrati: in questo modo, qualora si dovesse trovare un sottoproblema già risolto, si può usare la tabella per ricavarne velocemente la soluzione, invece di ricalcolarla. In maniera del tutto generale, si può descrivere la tecnica di programmazione dinamica nel modo seguente:

1. identifichiamo dei sottoproblemi del problema originario ed utilizziamo una tabella per memorizzare i risultati intermedi ottenuti;
2. all'inizio, definiamo i valori iniziali di alcuni elementi della tabella, corrispondenti ai sottoproblemi più semplici;
3. al generico passo, avanziamo in modo opportuno sulla tabella calcolando il valore della soluzione di un sottoproblema (corrispondente ad un dato elemento della tabella) in base alla soluzione dei sottoproblemi precedentemente risolti;
4. alla fine, restituiamo la soluzione del problema originario, che è stata memorizzata in un particolare elemento della tabella.

Esempio 9.1. Vogliamo risolvere il problema riguardante il calcolo della distanza tra due stringhe di caratteri: siano $X = x_1x_2...x_m$ e $Y = y_1y_2...y_n$ due stringhe di caratteri, rispettivamente di lunghezza m e n; possiamo definire il costo della trasformazione di X in Y come il numero di operazioni da apportare alla stringa X per ottenere Y. Le possibili operazioni che possiamo compiere sono:

- `inserisci(`a`)`: inserisci il carattere a nella posizione corrente della stringa;
- `cancella(`a`)`: cancella il carattere a dalla posizione corrente della stringa;
- `sostituisci(`a`, `b`)`: sostituisci il carattere a con il carattere b nella posizione corrente della stringa.

Assumendo che il costo di ciascuna operazione sia 1, possiamo definire il *costo della trasformazione tra X e Y* come la somma di tutti i costi pagati per compiere le operazioni di trasformazione da X in Y; la *distanza tra due stringhe X e Y* sarà il costo minimo di trasformazione di X in Y.

Indichiamo con $\delta(X, Y)$ la distanza tra le stringhe X e Y; inoltre, data una stringa X, definiamo il *prefisso di X fino al carattere i-esimo* come la stringa $X_i = x_1x_2...x_i$ se $1 \leq i \leq m$, la stringa vuota $X_0 = \emptyset$

se $i = 0$. Anziché risolvere il problema generale, proviamo a considerare i sottoproblemi $P(i, j)$, ovvero trovare la distanza $\delta(X_i, Y_j)$ tra il prefisso X_i e il prefisso Y_j:

- alcuni sottoproblemi sono particolarmente semplici: la soluzione del sottoproblema $P(0, j)$ è immediata, in quanto è sufficiente inserire, uno dopo l'altro, i j caratteri di Y_j e dunque si ha $\delta(X_0, Y_j) = j$; analogamente, la soluzione del sottoproblema $P(i, 0)$ consiste nel cancellare, uno dopo l'altro, gli i caratteri di X_i e dunque si ha $\delta(X_i, Y_0) = i$;
- il problema che vogliamo risolvere è il sottoproblema $P(m, n)$, ossia consiste nel calcolo della distanza $\delta(X_m, Y_n)$.

Ci manca solo un passo per poter scrivere l'algoritmo: come calcolare il valore della soluzione del sottoproblema $P(i, j)$ in funzione della soluzione dei sottoproblemi precedentemente risolti; chiaramente ci sono varie possibilità, a seconda dei valori di x_i e y_j. In particolare, se $x_i = y_j$, il costo minimo per trasformare X_i in Y_j è pari al costo minimo per trasformare X_{i-1} in Y_{j-1}; definendo D come la matrice in cui vengono salvati i risultati dei sottoproblemi, si ha $D[i, j] = D[i-1, j-1]$. Invece, se $x_i \neq y_j$, dobbiamo distinguere in base all'ultima operazione eseguita per trasformare X_i in Y_j:

- `inserisci(`y_j`)`: il costo minimo per la trasformazione è dato dal costo ottimo per trasformare X_i in Y_{j-1}, più 1 per l'inserimento del carattere y_j; si ha dunque:

$$D[i, j] = D[i, j-1] + 1$$

- `cancella(`x_i`)`: il costo minimo per la trasformazione è dato dal costo ottimo per trasformare X_{i-1} in Y_j, più 1 per la cancellazione del carattere x_i; si ha dunque:

$$D[i, j] = D[i-1, j] + 1$$

- `sostituisci(`x_i`,` y_j`)`: il costo minimo per la trasformazione è dato dal costo ottimo per trasformare X_{i-1} in Y_{j-1}, più per la sostituzione del carattere x_i con y_j; si ha dunque:

$$D[i, j] = D[i-1, j-1] + 1$$

Le precedenti relazioni assumono che sia nota l'ultima operazione utilizzata per la trasformazione di X_i in Y_j, anche se in realtà non è così; tuttavia sono possibili solo tre tipi di operazione e, dunque, dev'essere necessariamente usata una di esse: per trovare la scelta ottima, quindi, basta calcolare i valori relativi alle tre operazioni e scegliere il migliore. La relazione che lega la soluzione di un problema a quella di alcuni sottoproblemi è la seguente:

$$D[i, j] = \begin{cases} D[i-1, j-1] & \text{se } x_i = y_j \\ 1 + \min\{D[i, j-1], D[i-1, j], D[i-1, j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

La tabella delle distanze mantenuta dall'algoritmo, per la trasformazione della stringa `RISOTTO` nella stringa `PRESTO`, è la seguente:

		P	**R**	**E**	**S**	**T**	**O**
	0	**1**	2	3	4	5	6
R	1	1	**1**	2	3	4	5
I	2	2	2	**2**	3	4	5
S	3	3	3	3	**2**	3	4
O	4	4	4	4	**3**	3	3
T	5	5	5	5	**4**	**3**	4
T	6	6	6	6	5	**4**	4
O	7	7	7	7	6	5	**4**

L'algoritmo è il seguente:

algoritmo `distanzaStringhe` (*stringa X, stringa Y*) $\rightarrow$ *intero*

```
m = length(X), n = length(Y)
D: matrice[m + 1][n + 1]
for i = 0 to m do D[i, 0] = i
for j = 1 to n do D[0, j] = j
for i = 1 to m do
  for i = 1 to n do
    if(X[i] != Y[j]) then
      D[i, j] = 1 + min(D[i, j - 1], D[i - 1, j], D[i - 1, j - 1])
    else
      D[i, j] = D[i - 1, j - 1]
return D[m, n]
```

9.3 Paradigma greedy

Si tratta di una tecnica tipicamente usata per risolvere problemi di ottimizzazione; nella situazione più generale, in un problema di tale tipo avremo:

- un insieme di possibili candidati;
- l'insieme dei candidati già utilizzati;
- una funzione `ammissibile` che verifica se un insieme di candidati fornisce una soluzione (anche non ottima) al problema;
- una funzione `ottimo` che verifica se un insieme di candidati fornisce una soluzione ottima al problema;
- una funzione `seleziona` per estrarre un elemento dall'insieme dei candidati possibili non ancora esaminati;
- una funzione obiettivo che fornisce il valore di una soluzione.

Per risolvere il problema, occorre trovare un insieme di candidati che:

- è una soluzione;
- ottimizza il valore della funzione obiettivo.

Un algoritmo greedy generico può essere schematizzato come segue:

algoritmo `paradigmaGreedy` (*insieme di candidati C*) $\rightarrow$ *soluzione*

```
S = ∅
while(!ottimo(S) && (C != ∅)) do
  x = seleziona(C)
  C = C - {x}
  if(ammissibile(S ∪ {x}) then
    S = S ∪ {x}
if(ottimo(S)) then
  return S
else
  errore: non ho trovato soluzioni
```

Esempio 9.2. Consideriamo il problema del resto in un distributore automatico: vogliamo scrivere un algoritmo goloso per determinare l'insieme di monete da restituire per fare in modo che il numero di monete sia minore possibile; avremo:

- l'insieme dei candidati possibili è l'insieme delle monete del distributore;
- la funzione `ammissibile` restituisce vero se il valore delle monete nell'insieme scelto non è superiore al resto che dev0essere restituito;

- la funzione `ottimo` restituisce vero se il valore delle monete dell'insieme scelto è pari al resto da erogare;
- la funzione `seleziona` sceglie la moneta di valore maggiore tra quelle ancora da considerare;
- la funzione obiettivo restituisce il numero di monete della soluzione;
- la funzione `valore` restituisce il valore totale delle monete nell'insieme considerato.

Il codice è il seguente:

algoritmo `distribuisciResto` (*resto* R) $\rightarrow$ *soluzione*

```
1  C = monete nel distributore
2  S = ∅
3  while((valore(S) != R) && (C != ∅)) do
4    x = moneta di valore piu elevato in C
5    C = C - {x}
6    if(valore((S ∪ {x}) <= R) then
7      S = S ∪ {x}
8  if(valore(S) = R) then
9    return S come resto esatto
10 else
11   return S come resto parziale
```

Capitolo 10

Grafi

10.1 Definizioni preliminari

Definizione 10.1. Un *grafo G = (V, E)* consiste di:

- un insieme V di *vertici*;
- un insieme E di coppie di vertici, detti *archi*

Definizione 10.2. Un grafo si dice *non orientato* se E rappresenta una relazione *simmetrica* in V, ossia $(u, v) \in E \Leftrightarrow (v, u) \in E$, altrimenti si dice *orientato*; nel primo caso si ha $|E| \leq n \cdot (n-1)/2$, mentre nel secondo $|E| \leq n^2$.

Definizione 10.3. Dato un grafo $G = (V, E)$, $G' = (V', E')$ è *sottografo* di G se $V' \subseteq V$ e $E' \subseteq E \cap (V \times V')$.

Definizione 10.4. Dato un grafo $G = (V, E)$, $G' = (V', E')$ è *sottografo indotto* di G se $V' \subseteq V$ e $E' = E \cap (V \times V')$ (ossia, se contiene tutti e soli gli archi presenti in E i cui vertici appartengono entrambi a V').

Definizione 10.5. Dato $G = (V, E)$, un *cammino* in G è una sequenza di vertici $\pi = \langle x_0, x_1, ..., x_k \rangle$ tali che $(x_i, x_{i+1}) \in E, \forall i = 0..k-1$; la *lunghezza* del cammino è pari al numero di archi che lo compongono. π si dice *cammino semplice* se $i \neq j \Rightarrow x_i \neq x_j \forall i, j = 0..k$.

Definizione 10.6. Il vertice v si dice *raggiungibile* da u se esiste un cammino π tale che $x_0 = u$ e $x_k = v$.

Definizione 10.7. Dato un cammino π, un *sottocammino* è una sequenza di vertici $\langle x_i, x_{i+1}, ..., x_j \rangle$ con $0 \leq i \leq j \leq k$.

Definizione 10.8. Un *ciclo* è un cammino dove $x_0 = x_k$; un ciclo si dice *semplice* se tutti i vertici intermedi sono distinti.

Definizione 10.9. La *distanza* $\delta(u, v)$ tra due vertici u e v è il numero minimo di archi di un qualsiasi cammino da u a v; se v non è raggiungibile da u, allora $\delta(u, v) = \infty$.

Definizione 10.10. Se un cammino π da u a v è tale che $length(\pi) = \delta(u, v)$, allora π è un *cammino minimo.*

Definizione 10.11. Un grafo non orientato $G = (V, E)$ è *connesso* se, per ogni $u, v \in V$, esiste un cammino da u a v.

Definizione 10.12. Sia $G = (V, E)$ un grafo non orientato; $V' \subseteq V$ è *componente connessa* se:

- il sottografo indotto da V' è connesso;
- se $V'' \subseteq V$ induce un sottografo connesso e $V' \subseteq V''$, allora $V' = V''$ (*vincolo di massimalità*).

Definizione 10.13. Un *albero libero* è un grafo non orientato, connesso e aciclico.

Proprietà 10.1. Sia $G = (V, E)$ un albero libero, con $|V| = n$ e $|E| = m$: allora $m = n - 1$.

Dimostrazione. Procediamo per induzione su n.

Caso base: si ha $n = 1$, ossia l'albero con un nodo (e nessun arco); si ha $m = 0 = n - 1$.

Ipotesi induttiva: assumiamo valida la proprietà per $k < n$.

Passo induttivo: sia $n > 1$; prendiamo $v \in V$ e consideriamo il sottografo indotto da $V - \{v\}$, ossia $G' = (V', E')$ con $V' = V - \{v\}$ e $E' = E \cap (V' \times V')$. Siano $V_1, V_2, ..., V_k$ le componenti connesse di G'; si ha $n_1 + n_2 + ... + n_k = n - 1$, con n_i la cardinalità di V_i. V_i è connessa per definizione ed è aciclica, essendo sottografo indotto di un grafo aciclico; inoltre $|V_i| < n$, quindi posso usare l'ipotesi, ossia $m_i = n_i - 1$. Da ciascuna componente connessa ci dev'essere un arco che la congiunge a v, affinché il grafo di partenza sia un albero libero; si ha:

$$|E| = \sum_{1=1}^{k} m_i + k = \sum_{i=1}^{k} (n_i - 1) + k = \sum_{i=1}^{k} n_i - \sum_{i=1}^{k} 1 + k = n - 1 - k + k = n - 1$$

come volevasi dimostrare. □

Definizione 10.14. Sia $G = (V, E)$ un grafo non orientato: il *grado* di un vertice $u \in V$ è definito come

$$deg(u) = |\{v \in V : (u, v) \in E\}|$$

Sia ora $G = (V, E)$ un grafo orientato: il *grado entrante* e il *grado uscente* di un vertice $u \in V$ sono definiti, rispettivamente, come

$$deg_{in}(u) = |\{v \in V : (v, u) \in E\}|$$
$$deg_{out}(u) = |\{v \in V : (u, v) \in E\}|$$

Lemma 10.1. *Sia $G = (V, E)$ un grafo non orientato; si ha:*

$$\sum_{u \in V} deg(u) = 2 \cdot |E|$$

Sia ora $G = (V, E)$ un grafo orientato; si ha:

$$\sum_{u \in V} deg_{in}(u) = |E|$$
$$\sum_{u \in V} deg_{out}(u) = |E|$$

10.2 Rappresentazione di grafi

Una struttura dati di tipo grafo è descritta dal seguente schema generale:

Dati: un insieme di vertici (di tipo *vertice*) e di archi (di tipo *arco*).

Operazioni:

`numVertici()` → *intero*
Restituisce il numero di vertici presenti nel grafo.

`numArchi()` → *intero*
Restituisce il numero di archi presenti nel grafo.

`grado(`*vertice v*`)` → *intero*
Restituisce il numero di archi incidenti sul vertice v.

`archiIncidenti(`*vertice v*`)` → $\langle arco, arco, ..., arco \rangle$
Restituisce, uno dopo l'altro, gli archi incidenti sul vertice v.

`estremi(`*arco e*`)` → $\langle vertice, vertice \rangle$
Restituisce gli estremi x e y dell'arco $e = (x, y)$.

`opposto(`*vertice x, arco e*`)` → *vertice*
Restituisce y, l'estremo dell'arco $e = (x, y)$ diverso da x.

`sonoAdiacenti(`*vertice x, vertice y*`)` $\rightarrow$ *booleano*
Restituisce *true* se esiste l'arco (x, y), *false* altrimenti.

`aggiungiVertice(`*vertice v*`)` $\rightarrow$ *void*
Inserisce un nuovo vertice v.

`aggiungiArco(`*vertice x, vertice y*`)` $\rightarrow$ *void*
Inserisce un nuovo arco tra i vertici x e y.

`rimuoviVertice(`*vertice v*`)` $\rightarrow$ *void*
Cancella il vertice v e tutti gli archi ad esso incidenti.

`rimuoviArco(`*arco e*`)` $\rightarrow$ *void*
Cancella l'arco e.

10.2.1 Lista di archi

è una rappresentazione che si basa sull'utilizzo di:

- una struttura per rappresentare i vertici (tipo arraylist);
- una lista per rappresentare gli archi.

10.2.2 Liste di adiacenza

In questa rappresentazione, ogni vertice viene associato ad una lista contenente i suoi vertici adiacenti (struttura realizzata con array di liste).

10.2.3 Matrice di adiacenza

è una rappresentazione basata sull'uso di una matrice $n \times n$ le cui righe e colonne sono indicizzate dai vertici del grafo:

$$M[u,v] = \begin{cases} 1 & \text{se } (u,v) \in E \\ 0 & \text{altrimenti} \end{cases}$$

Si tratta di una rappresentazione utile per il seguente fatto:

Proprietà 10.2. Sia $M_k = \underbrace{M \cdot M \cdot ... \cdot M}_{kvolte}$: si ha $M^k[u,v] = 1$ se e solo se esiste un cammino di k vertici che collega u a v.

10.2.4 Confronto tra le rappresentazioni

Operazione	Lista di archi	Lista di adiacenza	Matrice di adiacenza
`grado(`v`)`	$O(m)$	$O(deg(v))$	$O(n)$
`archiIncidenti(`v`)`	$O(m)$	$O(deg(v))$	$O(n)$
`sonoAdiacenti(`x, y`)`	$O(m)$	$O(\min\{deg(x), deg(y)\})$	$O(1)$
`aggiungiVertice(`v`)`	$O(1)$	$O(1)$	$O(n^2)$
`aggiungiArco(`x, y`)`	$O(1)$	$O(1)$	$O(1)$
`rimuoviVertice(`v`)`	$O(m)$	$O(m)$	$O(n^2)$
`rimuoviArco(`e`)`	$O(m)$	$O(deg(x) + deg(y))$	$O(1)$

Tabella 10.1: Tempi di esecuzione

	Lista di archi	Lista di adiacenza	Matrice di adiacenza
Spazio	$O(m+n)$	$O(m+n)$	$O(n^2)$

Tabella 10.2: Spazio occupato

10.3 Visite di grafi

Visitare un grafo significa visitarne tutti i nodi, assicurandosi di passare per ciascuno *una ed una sola volta*, nell'ipotesi che il grafo sia connesso, partendo da un nodo *sorgente* s. Durante la visita, i nodi vengono *marcati* con opportuni *colori* per tenere traccia del fatto che siano stati già considerati o meno:

Verde: nodo non ancora considerato.

Giallo: nodo già considerato, ma non ancora visitato.

Rosso: nodo già visitato.

L'algoritmo costruisce un sottografo T di G i cui archi formano un albero radicato in s; mantiene, inoltre, una struttura U in cui inserisce i vertici da esplorare.

algoritmo `visitaGenerica` (*grafo G, vertice s*) → *albero*

```
1  for each u ∈ V do
2    color[u] = VERDE
3  color[s] = GIALLO
4  inizializza U e T inserendo in entrambi S
5  while U != ∅ do
6    estrai un vertice u da U
7    color[u] = ROSSO
8    visita u
9    for each v ∈ Adj[u] do
10     if(color[v] == VERDE) then
11       color[v] = GIALLO
12       inserisci v in U e in T ricorda (u, v)
13 return T
```

Proprietà 10.3.

- durante il ciclo while, U contiene solo vertici *gialli*, $T - U$ solo vertici *rossi* e $V - T$ solo vertici *verdi*;
- ogni vertice raggiungibile da s viene inserito in U;
- ogni vertice viene colorato *esattamente una volta* per ogni colore e nell'ordine verde, giallo, rosso:
  - → ogni vertice viene inserito in U esattamente una volta;
  - → il corpo del while viene eseguito *una volta* per ogni vertice;
- se il corpo del while viene eseguito sul vertice v, ha lo stesso costo asintotico della ricerca dei vertici che sono in `Adj[v]`:
  - → costo del while = $\sum_{v \in V}$ costo della ricerca dei vertici in `Adj[v]`

Complessità: il costo totale è dato da 'costo del primo for each' + 'costo del while'

Lista di archi

$$\text{costo while} = \sum_{v \in V} O(m) = O(m \cdot n)$$
$$\text{costo totale} = O(n + m \cdot n) = O(m \cdot n)$$

Liste di adiacenza

$$\text{costo while} = \sum_{v \in V} O(deg(v)) = O(\sum_{v \in V} deg(v)) = O(2 \cdot m) = O(m)$$
$$\text{costo totale} = O(n + m)$$

Matrice di adiacenza

$$\text{costo while} = \sum_{v \in V} O(n) = O(\sum_{v \in V} n) = O(n^2)$$
$$\text{costo totale} = O(n + n^2) = O(n^2)$$

Se U è gestito come:

- *pila*, viene eseguita una *visita in profondità*;
- *coda*, viene eseguita una *visita in ampiezza*.

Proprietà 10.4. Sia T l'albero prodotto dalla visita in ampiezza del grafo G: allora, per ogni nodo v, il livello di v in T è pari alla distanza tra la sorgente s e il nodo stesso; per tale ragione, T si dice *albero dei cammini minimi*.

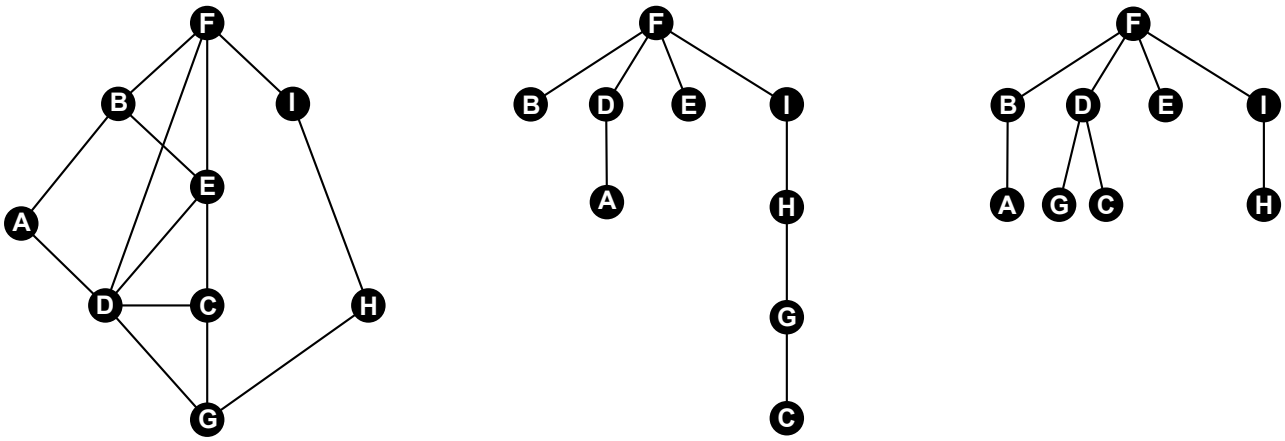


Figura 10.1: Grafo, albero della visita in profondità e albero della visita in ampiezza partendo da F

Capitolo 11

Minimo albero ricoprente

Definizione 11.1. Dato un grafo $G = (V, E)$ non orientato e connesso, un albero ricoprente di G è un sottografo $T \subseteq E$ tale che:

- T è un albero;
- T contiene tutti i vertici di G.

Sia definita una *funzione peso* $w : E \to \mathbb{R}$; possiamo definire il costo di un albero di copertura come la somma dei pesi degli archi che lo compongono:

$$w(T) = \sum_{e \in T} w(e)$$

Definizione 11.2. Dato un grafo $G = (V, E)$ non orientato, connesso e pesato sugli archi, un *minimo albero ricoprente* di G è un albero di copertura di G di costo minimo.

Le definizioni viste finora possono essere estese al caso in cui G non sia connesso: in tal caso si parlerà di *minima foresta ricoprente*.

Lemma 11.1. *Siano $G = (V, E)$ un grafo connesso e non orientato, $T \subseteq E$ un albero di copertura, $(u, v) \in E - T$ e p un cammino semplice da u a v con tutti gli archi in T (esiste in quanto T è connesso); allora $(T - \{(x, y)\}) \cup \{(u, v)\}$ è un albero di copertura.*

Dimostrazione. Sia $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$: poiché $(u, v) \notin T$, si ha che (u, v) non fa parte del cammino p, ossia che $(u, v) \neq (x, y)$, da cui $|T'| = |V| - 1$. Ci rimane da provare che T' è connesso. Consideriamo il cammino $p = p_1, \langle x, y \rangle, p_2$, dove:

- p_1 è un cammino da u a x che non contiene (x, y);
- p_2 è un cammino da y a v che non contiene (x, y).

Siano $a, b \in V$: dobbiamo trovare un cammino da a a b fatto di archi in T'. Poiché T è connesso, esiste un cammino q da a a b fatto da archi in T; possiamo avere due casi:

1. se q non contiene l'arco (x, y), è un cammino anche in $T - \{(x, y)\}$ e dunque anche in T';
2. se q contiene l'arco (x, y), possiamo scrivere $q = q_1, \langle x, y \rangle, q_2$ dove:
   - q_1 è un cammino da a a x che non contiene (x, y);
   - q_2 è un cammino da y a b che non contiene (x, y).

   Sia $q' = q_1, \overleftarrow{p_1}, \langle u, v \rangle, \overleftarrow{p_2}, q_2$: q' è cammino in T', dunque a e b sono connessi in T'.

Abbiamo che T' è connesso e, avendo $n - 1$ archi, è anche aciclico: dunque è un albero di copertura, come volevasi dimostrare. □

11.1 Teorema fondamentale

Definizione 11.3. Un *taglio* $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V.

Definizione 11.4. Un arco *attraversa* il taglio $(S, V - S)$ se uno dei suoi estremi si trova in S e l'altro in $V - S$.

Definizione 11.5. Un taglio *rispetta* un insieme A di archi se nessun arco di A attraversa il taglio.

Definizione 11.6. Un arco che attraversa un taglio si dice *leggero* se ha peso pari al minimo tra i pesi di tutti gli archi che attraversano tale taglio.

Definizione 11.7. Sia A sottoinsieme di un qualche albero di copertura minimo; un arco si dice *sicuro* per A se può essere aggiunto ad A e quest'ultimo continua ad essere sottoinsieme di un albero di copertura minimo.

Teorema 11.1. *Sia $G = (V, E, w)$ un grafo non orientato, connesso e pesato; sia inoltre:*

- *$A \subseteq E$ contenuto in qualche MST;*
- *$(S, V - S)$ un taglio che rispetta A;*
- *$(u, v) \in E$ un arco leggero che attraversa il taglio.*

Allora (u, v) è sicuro per A.

Dimostrazione.

Ipotesi: esiste $T \subseteq E$ MST tale che $A \subseteq T$.

Tesi: dobbiamo trovare $T' \subseteq E$ MST tale che $A \cup \{(u, v)\} \subseteq T'$.

Poiché $(S, V - S)$ rispetta A e (u, v) attraversa il taglio, allora $(u, v) \notin A$. Possiamo distinguere due casi:

1. se $(u, v) \in T$, allora $A \cup \{(u, v)\} \subseteq T$ MST.
2. se $(u, v) \notin T$, dal momento che quest'ultimo è connesso, esisterà in esso un cammino semplice p da u a v; poiché (u, v) attraversa il taglio, esiste almeno un arco (x, y) di p che lo attraversa. Sia $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$: per il lemma precedentemente dimostrato, è un albero di copertura. Si ha:
   (a) $(u, v) \in T'$;
   (b) $A \subseteq T$ e, poiché (x, y) attraversa il taglio ed il taglio rispetta A, $(x, y) \notin A$; da questo segue:
   $$A \subseteq T - \{(x, y)\} \rightarrow A \subseteq (T - \{(x, y)\}) \cup \{(u, v)\} = T' \rightarrow A \cup \{(u, v)\} \subseteq T'$$
   T' è un albero di copertura contenente $A \cup \{(u, v)\}$.
   (c) $w(T') = w(T) - w(x, y) + w(u, v)$; dal momento che (x, y) attraversa il taglio e (u, v) attraversa il taglio ed è leggero:
   $$w(x, y) \geq w(u, v) \rightarrow w(u, v) - w(x, y) \leq 0 \rightarrow w(T') \leq w(T)$$
   Dal momento che T è MST per ipotesi, segue che $w(T') = w(T)$; dunque anche T' è MST e contiene $A \cup \{(u, v)\}$. □

Corollario 11.1. *Sia $G = (V, E, w)$ un grafo non orientato, connesso e pesato; siano:*

- *$A \subseteq E$ contenuto in un MST;*
- *C componente connessa della foresta $G_A = (V, A)$;*
- *$(u, v) \in E$ arco leggero che connette C ad un'altra componente connessa di G_A.*

Allora (u, v) è sicuro per A.

Dimostrazione. è sufficiente applicare il teorema fondamentale considerando il taglio $(C, V - C)$; poiché:

- il taglio rispetta A;
- (u, v) è leggero per il taglio;

si hanno le ipotesi del teorema, dal quale si ottiene che (u, v) è sicuro per A. □

11.2 Algoritmo di Kruskal

L'algoritmo mantiene, istante per istante, una *foresta* contenuta in qualche MST; per poter gestire gli insiemi disgiunti che rappresentano le varie componenti connesse della foresta, occorre usare una struttura dati appropriata. La tecnica utilizzata consiste nel partizionare l'insieme V in k classi tali che:

- $\bigcup_{i=1}^{k} V_i = V$
- $i \neq j \Rightarrow V_i \cap V_j = \emptyset$

e rappppresentare ogni classe con un elemento della classe stessa; devono essere permesse le seguenti operazioni:

`makeSet(v)`: crea una classe il cui unico elemento è v e lo elegge come rappresentante della stessa (costo $O(1)$);

`findSet(v)`: restituisce il rappresentante della classe cui v appartiene (costo $O(1)$);

`union(u, v)`: unisce le classi di u e v (partendo da una partizione generata da n chiamate a `makeSet`, se si eseguono k `union`, il costo totale di tutte le operazioni è $O(k \cdot \log k)$).

algoritmo `mstKruskal` (*grafo pesato* G) $\rightarrow$ *MST*

```
1 A = ∅
2 for each u ∈ V do
3   makeSet(u)
4 ordina gli archi di E in modo non decrescente rispetto al peso
5 for each (u, v) ∈ E do
6   if(findSet(u) != findSet(v)) then
7     A = A ∪ {(u, v)}
8     union(u, v)
9 return A
```

Invariante di ciclo: $A \subseteq MST$.

Complessità

1. $O(1)$

2-3. $O(n)$

4. $O(m \cdot \log m)$

5-8. in tutte le iterazioni, si eseguono in totale:

   6. 2 `findSet` per ogni arco $\rightarrow O(2 \cdot m)$
   7. A viene aggiornato $n-1$ volte (diventa un albero) $\rightarrow O(n)$
   8. $n-1$ `union` $\rightarrow O(n \cdot \log n)$

Totale: $O(1) + 2 \cdot O(n) + O(m \cdot \log m) + O(2 \cdot m) + O(n \cdot \log n) = O(m \cdot \log m) = O(m \cdot \log n)$, poiché $m = O(n^2)$.

Esempio di esecuzione

Assumiamo che gli archi siano ordinati nel modo seguente: *AB, CD, BD, BC, AD, AC*. Consideriamo, nell'ordine, gli archi che vengono selezionati:

1. AB: poiché i nodi A e B non sono ancora collegati, l'arco viene selezionato;
2. CD: non esiste un cammino tra C e D, dunque l'arco viene selezionato;
3. BD: non esiste un cammino tra B e D, quindi l'arco viene scelto (il MST è ora pronto);
4. CD: esiste già un cammino tra C e D, dunque l'arco non viene selezionato;
5. AD: è già presente un cammino tra A e D, perciò l'arco viene scartato;
6. AC: tra A e C esiste già un cammino, quindi l'arco non viene scelto.

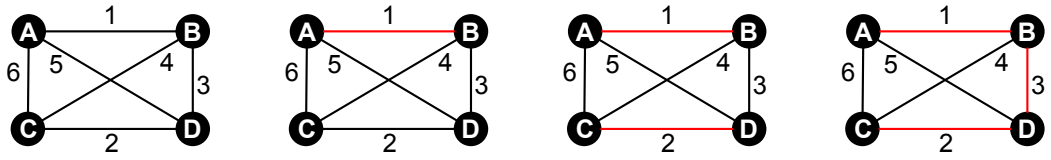


Figura 11.1: Simulazione dell'esecuzione dell'algoritmo di Kruskal

11.3 Algoritmo di Prim

L'algoritmo di Prim è fondato direttamente sul teorema fondamentale degli alberi ricoprenti minimi e mantiene, istante per istante, un *albero* contenuto in qualche MST; l'algoritmo, inoltre, vuole in ingresso anche un nodo $r \in V$ che sarà la radice dell'albero di copertura.
Viene utilizzata una struttura dati del tipo *coda a min-priorità* per gestire l'insieme dei vertici Q non ancora inclusi nell'albero; per ciascun vertice $u \in Q$ vengono memorizzati i seguenti attributi:

- `key[u]` è la chiave di priorità: il suo valore è pari al minimo tra i pesi di tutti gli archi che collegano u ai nodi dell'albero $V - Q$, se u è adiacente a un vertice di tale albero, altrimenti il valore è infinito.
- π`[u]` memorizza il predecessore di u nell'albero generato dall'algoritmo.

Si può ricostruire l'albero finale utilizzando i nodi e le informazioni sui predecessori:

$$A = \{(u, \pi[u]) \in E : u \in (V - \{r\})\}$$

algoritmo `mstPrim` (*grafo pesato G, radice r*) $\rightarrow$ *MST*

```
1  Q: coda di priorita contenente tutti i vertici in V
2  for each u ∈ Q do
3    key[u] = ∞
4  key[r] = 0
5  π[r] = NIL
6  while(Q != ∅) do
7    u = extractMin(Q)
8    for each v ∈ Adj[u] do
9      if(v ∈ Q && w(u, v) < key[v]) then
10       key[v] = w(u, v)
11       π[v] = u
12 return A = {(u, π[u]) ∈ E : u ∈ (V - {r})}
```

Correttezza: segue direttamente dal teorema fondamentale (ad ogni iterazione, basta considerare il taglio $(V - Q, Q)$.

Complessità

1-5. $O(n)$

7. n estrazioni di costo $O(\log n) \rightarrow O(n \cdot \log n)$

8-11. consideriamo tutti i vertici adiacenti a quello estratto: l'operazione più costosa è il decremento della chiave, del costo $O(\log n)$: per ogni vertice $u \in Q$, si paga:

$$\sum_{v \in Adj[u]} O(\log n) = deg(u) \cdot O(\log n)$$

Considerando tutti i vertici, si ha:

$$\sum_{u \in V} \delta(u) \cdot O(\log n) = 2 \cdot m \cdot O(\log n) = O(m \cdot \log n)$$

12. costo della ricostruzione $O(n)$

Totale: $O(n) + O(n \cdot \log n) + O(m \cdot \log n) + O(n) = O(m \cdot \log n)$, poiché $m \geq n - 1$.

Esempio di esecuzione

Per ciascun nodo, è indicata la coppia (*chiave, predecessore*) associata ad ogni iterazione del ciclo; i nodi marcati in nero sono quelli ancora presenti in Q, il nodo rosso è quello estratto nell'iterazione considerata e quelli blu sono i nodi già estratti da Q ed esaminati.

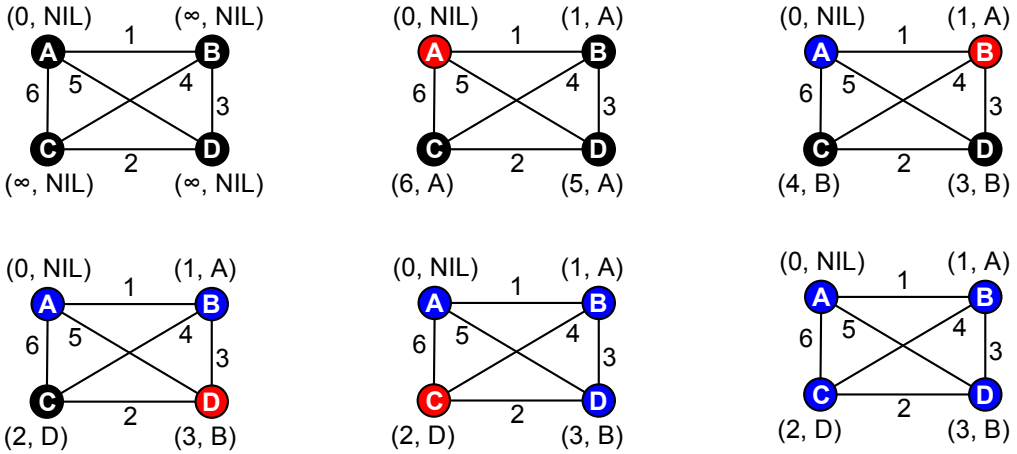


Figura 11.2: Simulazione dell'esecuzione dell'algoritmo di Prim (nodo sorgente a)

L'albero restituito dall'algoritmo, ricostruito a partire dai dati dell'ultima figura, è quello costituito dagli archi *AB, BD* e *CD*.

Capitolo 12

Cammini minimi con sorgente singola

Definizione 12.1. Sia $G = (V, E, w)$ un grafo orientato e pesato; dato il cammino $p = \langle v_0, v_1, ..., v_k \rangle$ in G, il valore $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$ rappresenta il *peso* del cammino.

Dati $u, v \in V$, definiamo:

$$\mathcal{C}(u, v) = \{p : p \text{ è cammino da } u \text{ a } v \text{ in } G\}$$

L'insieme è infinito se il grafo è ciclico (infatti basta eseguire più volte un ciclo per ottenere cammini diversi).

Definizione 12.2. La *distanza* da u a v corrisponde al peso minimo di un cammino da u a v:

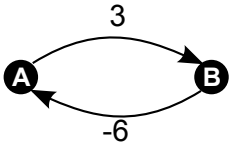
$$\delta(u, v) = \begin{cases} +\infty & \text{se } \mathcal{C}(u, v) = \emptyset \\ \min\limits_{p \in \mathcal{C}(u,v)} w(p) & \text{se } \mathcal{C}(u, v) \neq \emptyset \end{cases}$$

Definizione 12.3. Sia $p = \langle u, ..., v \rangle$: si dice che p è *cammino minimo* se $w(p) = \delta(u, v)$.

Varianti al problema dei cammini minimi

- cammino minimo tra due vertici;
- cammini minimi con sorgente singola;
- cammini minimi con destinazione singola;
- cammini minimi tra tutte le coppie di vertici.

Cicli di peso negativo



L'esistenza dei cicli di peso negativo è problematica nell'ambito della ricerca dei cammini minimi; nell'esempio in figura, *non esiste* un cammino minimo tra a e b: infatti, se supponiamo di aver trovato un cammino minimo, basta eseguire un'altra volta il ciclo e se ne ottiene uno di peso inferiore.

Rappresentazione dei cammini minimi

Ad ogni vertice, associamo un attributo π`[u]`, che rappresenta il precedente di u in un albero di cammini minimi; gli algoritmi aggiornano opportunamente i valori di π in modo che, alla fine, l'albero rappresentato sia un albero di cammini minimi. Definiamo:

Sottografo dei predecessori indotto da π
$G_\pi = (V_\pi, E_\pi)$
$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$
$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$

Albero dei cammini minimi: dato $G = (V, E, w)$ orientato e pesato, l'albero dei cammini minimi è $G' = (V', E')$, con $V' \subseteq V$, $E' \subseteq E$ tale che:

- V' è l'insieme dei vertici raggiungibili da s in G;
- G' è albero radicato in s;
- $\forall u \in V'$, l'unico cammino da s a u in G' è un cammino minimo da s a u in G.

12.1 Cammini minimi e rilassamento

Lemma 12.1 (Sottocammini di cammini minimi sono cammini minimi)**.** *Siano $G = (V, E, w)$ un grafo orientato e pesato e $p = \langle v_0, v_1, .., v_k \rangle$ cammino minimo da v_0 a v_k; presi gli indici $0 \leq i < j \leq k$, il sottocammino $p_{ij} = \langle v_i, ..., v_j \rangle$ è cammino minimo da v_i a v_j.*

Dimostrazione. Procediamo per assurdo. Assumiamo che p_{ij} non sia minimo: esisterà dunque un cammino q da v_i a v_j tale che $w(q) < w(p_{ij})$; consideriamo ora il cammino p_{0i}, q, p_{jk}:

$$\begin{aligned} w(p_{0i}, q, p_{jk}) &= w(p_{0i}) + w(q) + w(p_{jk}) \\ &< w(p_{0i}) + w(p_{ij}) + w(p_{jk}) \\ &= w(p) \end{aligned}$$

Ma p è cammino minimo per ipotesi: *assurdo.* Quindi p_{ij} deve essere minimo. □

Proposizione 12.1. *Siano $G = (V, E, w)$ grafo orientato e pesato e $p = \langle v_0, ..., v_k \rangle$ cammino minimo da $u = v_0$ a $v = v_k$; allora $\forall i = 0..k, \delta(u, v) = \delta(u, v_i) + \delta(v_i, v)$.*

Dimostrazione. Segue direttamente dal lemma precedente; sia $p = p_{0i}, p_{ik}$:

$$\begin{aligned} w(p) &= w(p_{0i}) + w(p_{ik}) \\ \delta(u, v) &= \delta(u, v_i) + \delta(v_i, v) \end{aligned}$$

come si voleva dimostrare. □

Corollario 12.1. *Siano $G = (V, E, w)$ un grafo orientato e pesato e p un cammino da s a v scomponibile in un cammino da s a $u \in V$, di nome p' e un arco da u a v; allora il peso di p è pari a $\delta(s, v) = \delta(s, u) + w(u, v)$.*

Lemma 12.2. *Siano $G = (V, E, w)$ un grafo orientato e pesato e $s \in V$ un nodo sorgente; allora, $\forall (u, v) \in E$ si ha $\delta(s, v) \leq \delta(s, u) + w(u, v)$.*

Dimostrazione. Dobbiamo distinguere due casi:

- u non è raggiungibile da s:

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

- u è raggiungibile da s, dunque esiste un cammino p da s a u; sia q il cammino formato da p e (u, v):

$$\begin{aligned} \delta(s, v) &\leq w(q) \\ &= w(p) + w(u, v) \\ &= \delta(s, u) + w(u, v) \end{aligned}$$

In entrambi i casi la proprietà è valida, dunque il lemma è dimostrato. □

Algoritmi

Gli algoritmi che analizzeremo (Dijkstra, Bellman-Ford) aggiornano due attributi assegnati ai nodi:

`d[u]` : stima del peso del cammino minimo dalla sorgente a u;

π`[u]` : predecessore di u nell'albero dei predecessori.

Al termine vogliamo avere:

- $d[u] = \delta(s, u)$;
- G_π è l'albero dei cammini minimi.

Inizializzazione

funzione `initSingleSource` (*grafo pesato G, sorgente S*) $\rightarrow$ *void*

```
1 for each u ∈ V do
2   d[u] = +∞
3   π[u] = NIL
4 d[s] = 0
```

Rilassamento

funzione `relax` (*nodo u, nodo v, funzione peso w*) $\rightarrow$ *void*

```
1 /* Precondizione: (u, v) in E */
2 if(d[v] > d[u] + w(u, v)) then
3   d[v] = d[u] + w(u, v)
4   π[v] = u
```

Lemma 12.3. *Dopo la chiamata a* ***relax****(u, v, w) si ha che $d[v] \leq d[u] + w(u, v)$, qualunque sia l'ordine delle chiamate alla funzione.*

Lemma 12.4. *Siano $G = (V, E, w)$ un grafo orientato e pesato e $s \in V$ il nodo sorgente; supponiamo che all'inizio sia stata invocata* ***initSingleSource****(G, s). Allora:*

1. *$\forall u \in V$ si ha $d[u] \geq \delta(s, u)$;*
2. *la proprietà al punto 1 rimarrà invariata per qualunque sequenza di* ***relax****;*
3. *nel momento in cui si dovesse verificare che $d[u] = \delta(s, u)$ per qualche $u \in V$, allora $d[u]$ non verrà più modificato da alcuna chiamata di* ***relax****.*

Dimostrazione.

1. Dopo `initSingleSource(`G, s`)`:
   - se $u \neq s, d[u] = +\infty \geq \delta(s, u)$
   - se $u = s, d[u] = 0$
     - se s non sta in un ciclo negativo:
       $$\delta(s, s) = 0$$
       $$d[s] = \delta(s, s)$$
     - se s sta in un ciclo negativo:
       $$\delta(s, s) = -\infty$$
       $$d[s] = 0 \leq \delta(s, s)$$

2. Procediamo per assurdo. Assumiamo che, dopo un certo numero di `relax`, ci sia un vertice v tale che $d[v] < \delta(s, v)$. Supponiamo, inoltre, che v sia il *primo* vertice per cui valga questa proprietà, immediatamente dopo la chiamata `relax`(u, v, w); abbiamo:

$$d[v] < d[u] + w(u, v) < \delta(u, v) \leq \delta(s, u) + w(u, v)$$

e, per la proprietà transitiva:

$$\begin{aligned} d[u] + w(u, v) &< \delta(s, u) + w(u, v) \\ d[u] &< \delta(s, u) \end{aligned}$$

Visto che $d[u]$ non può essere stato modificato da `relax`(u, v, w), dev'essere stato cambiato prima da un'altra chiamata; ma v è stato definito come il primo vertice tale per cui $d[v] < \delta(s, v)$: *assurdo.*

3. Osserviamo che:

   - nessuna `relax`, per definizione, incrementa $d[u]$, ma può solo decrementarlo;
   - $d[u] \geq \delta(s, u)$ dopo ogni `relax`.

   Segue che, dal momento in cui $d[u] = \delta(s, u)$, tale valore non può più cambiare. □

Corollario 12.2. *Se G è inizializzato con `initSingleSource`(G, s) e v **non** è raggiungibile da s, allora $d[v] = +\infty$ non verrà mai aggiornato da alcuna `relax`.*

Lemma 12.5. *Siano dati $G = (V, E, w)$ grafo orientato e pesato e p il cammino minimo da s a v che include (u, v) come arco finale. Supponiamo di inizializzare con `initSingleSource`(G, s) e chiamiamo `relax` più volte; se, ad un certo punto, si ha $d[u] = \delta(s, u)$, allora dopo la chiamata `relax`(u, v, w) si ha $d[v] = \delta(s, v)$.*

Dimostrazione. Prima di `relax`(u, v, w) vale $\delta(s, v) \leq d[v]$ per il lemma 12.4 e che $d[u] = \delta(s, u)$ per ipotesi. Dopo `relax`(u, v, w) si ha che:

$$\begin{aligned} d[v] &\leq d[u] + w(u, v) \\ &= \underbrace{\delta(s, u)}_{w(p')} + w(u, v) \\ &= w(p') + w(u, v) \\ &= w(p) \\ &= \delta(s, v) \end{aligned}$$

dove p' è cammino minimo da s a u essendo sottocammino di p. Dal lemma 12.4 si ha che $\delta(s, v) \leq d[v]$, da cui segue $d[v] = \delta(s, v)$. □

12.2 Algoritmo di Dijkstra

algoritmo `dijkstra` (*grafo pesato G, sorgente s*) → *albero dei cammini minimi*

```
1  /* Precondizione: pesi strettamente non negativi */
2  initSingleSource(G, s)
3  S = ∅
4  Q = V
5  while(Q != ∅) do
6    u = extractMin(Q)
7    S = S ∪ {u}
8    for each v ∈ Adj[u] do
9      relax(u, v, w)
10 return (d, G_π)
```

Complessità

Sia Q implementato come heap binario:

1. $O(n)$
2. $O(1)$
3. $O(n)$

4-8. il ciclo viene eseguito n volte:

   5. ogni estrazione costa $O(\log n)$, per un totale di $O(n \cdot \log n)$
   6. $O(1)$, per un totale di $O(n)$
   7-8. chiama `relax` una volta per ogni arco del grafo, quindi si ha un costo totale di $O(m \cdot \log n)$; ogni chiamata, infatti, può modificare la chiave di priorità, operazione dal costo $O(\log n)$

Totale: $3 \cdot O(n) + O(1) + O(n \cdot \log n) + O(m \cdot \log n) = O((n + m) \cdot \log n)$.

Sia Q implementato come array lineare:

1. $O(n)$
2. $O(1)$
3. $O(n)$

4-8. il ciclo viene eseguito n volte:

   5. bisogna scorrere ogni volta l'intero array, per un totale di $O(n^2)$
   6. $O(1)$, per un totale di $O(n)$
   7-8. chiama `relax` una volta per ogni arco del grafo, che ha costo costante; in totale, dunque, il costo è $O(m)$

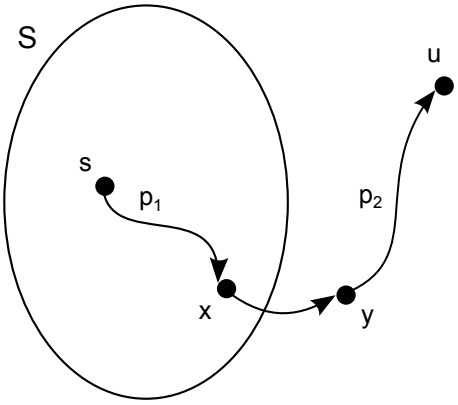
Totale: $3 \cdot O(n) + O(1) + O(n^2) + O(m) = O(n^2 + m)$.
La prima organizzazione risulta migliore nel caso il grafo sia *sparso* ($m = O(n)$), mentre la seconda è conveniente se il grafo è denso ($m = O(n^2)$).

Correttezza

Teorema 12.1. *Sia $G = (V, E, w)$ un grafo orientato e pesato con $w(u, v) \geq 0, \forall (u, v) \in E$. Allora, al termine dell'esecuzione dell'algoritmo, si ha:*

1. *$d[u] = \delta(s, u), \forall u \in V$;*
2. *G_π è l'albero dei cammini minimi.*

Dimostrazione. Dimostriamo solo il primo punto, in quanto il secondo segue da `relax`.
Proviamo, comunque, una proprietà più forte: $\forall u \in V$, nel momento in cui u è inserito in S, vale $d[u] = \delta(s, u)$ (e per il lemma 12.4, non diminuirà ulteriormente).
Procediamo per assurdo: assumiamo che u sia il primo vertice tale che $d[u] \neq \delta(s, u)$ nel momento in cui viene inserito in S. Dopo `initSingleSource`(G, s) si ha $d[s] = 0$; nel grafo non ci sono cicli negativi, poiché i pesi sono tutti non negativi, dunque si ha $\delta(s, s) = 0$: abbiamo dunque $d[s] = \delta(s, s)$, quindi il vertice u non può essere s.
Visto che s viene estratto per primo, $S \neq \emptyset$ quando u viene inserito; può essere $\delta(s, u) = +\infty$? Dopo `initSingleSource`, $d[u] = +\infty$, quindi ancora prima di iniziare le iterazioni si ha $d[u] = \delta(s, u)$; nelle iterazioni si eseguono delle chiamate a `relax` che, però, non modificano $d[u]$: segue, dunque, che u dev'essere raggiungibile da s, ossia $\delta(s, u) < +\infty$. Sia p il cammino minimo da s a u; la situazione che si ha quando viene estratto u da Q è:



$$Q = V - S$$

Sia (x, y) arco di p tale che $x \in S$ e $y \in Q$. Si ha:

$d[u] \leq d[y]$, perché u estratto da Q che contiene anche y;

$$d[u] = \min_{v \in Q} d[v]$$

$d[x] = \delta(s, x)$ perché u è il primo vertice per cui non vale $d[u] = \delta(s, u)$ e x è inserito in S prima di u. Inoltre, quando è stato inserito x, è stata chiamata `relax`(x, y, w); dopo questa chiamata, $d[y] = \delta(s, y)$ per il lemma 12.5.

$\delta(s, y) \leq \delta(s, u)$ poiché i pesi sono non negativi.

$\delta(s, u) \neq d[u]$ per ipotesi e $d[u] \geq \delta(s, u)$ per il lemma 12.4; segue che $d[u] > \delta(s, u)$.

Da tutte queste proprietà, segue:

$$\begin{aligned} d[u] &\leq d[y] \\ &= \delta(s, y) \\ &\leq \delta(s, u) \\ &< d[u] \end{aligned}$$

Si ottiene dunque un *assurdo*: il vertice u che cercavamo non esiste e la correttezza è dimostrata.

Esempio di esecuzione

Per ciascun nodo, è indicata la coppia (*chiave, predecessore*) associata ad ogni iterazione del ciclo; i nodi marcati in nero sono quelli ancora presenti in Q, il nodo rosso è quello estratto nell'iterazione considerata e quelli blu sono i nodi già estratti da Q ed esaminati. In ogni figura, sono evidenziati gli archi che compongono il sottografo indotto da π.

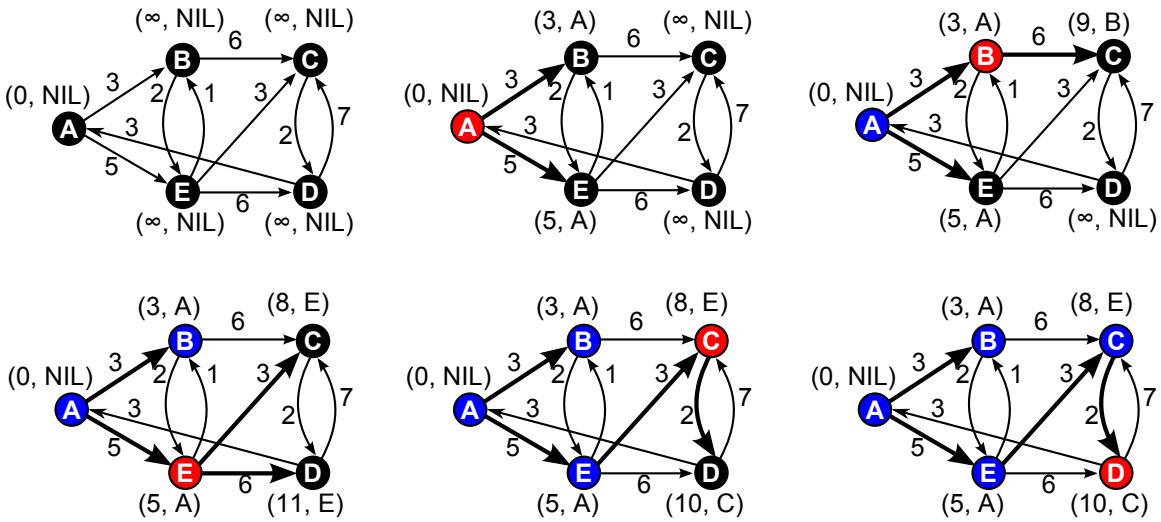


Figura 12.1: Simulazione dell'esecuzione dell'algoritmo di Dijkstra (nodo sorgente a)

12.3 Algoritmo di Bellman-Ford

L'algoritmo di Dijkstra presenta dei problemi nel caso in cui siano presenti dei cicli di peso negativo: per risolverli, è stato proposto l'algoritmo di *Bellman-Ford*.

algoritmo `bellmanFord` (*grafo pesato G, sorgente s*) → *booleano*

```
1 initSingleSource(G, s)
2 for i = 1 to |V| - 1 do
3   for each (u, v) ∈ E do
4     relax(u, v, w)
5 for each (u, v) ∈ E do
6   if(d[v] > d[u] + w(u, v)) then
7     return false
8 return true
```

L'algoritmo restituisce:

- *false*, se è stato trovato un ciclo negativo;
- *true* altrimenti.

Osservazione: l'algoritmo ritorna *true* se e solo se $d[v] \leq d[u] + w(u,v), \forall (u,v) \in E$.

Correttezza

Teorema 12.2. *Siano $G = (V, E, w)$ un grafo orientato e pesato e $s \in V$ il nodo sorgente:*

1. *se il grafo non contiene cicli negativi raggiungibili da s allora, alla fine di `bellmanFord(G, s)` si ha:*
   (a) *per ogni $u \in V$ risulta $d[u] = \delta(s,u)$;*
   (b) *G_π è un albero di cammini minimi;*
   (c) *l'algoritmo ritorna true;*
2. *se il grafo G contiene cicli negativi raggiungibili da s, l'algoritmo ritorna false.*

Dimostrazione.

Punto 1a. Per ipotesi, G non ha cicli negativi raggiungibili da s. Sia $u \in V$; dobbiamo provare che vale $d[u] = \delta(s,u)$ al termine dell'esecuzione dell'algoritmo:

1. se u non è raggiungibile da s, si ha $\delta(s,u) = +\infty$. Dopo `initSingleSource(`G, s`)` vale $d[u] = +\infty = \delta(s,u)$ e, poiché `relax` non modifica ulteriormente $d[u]$ per il lemma 12.4, al termine vale ancora $d[u] = \delta(s,u)$;
2. se u è raggiungibile da s, si ha $\delta(s,u) < +\infty$. Sia $p = \langle x_0, ..., x_k \rangle$ cammino minimo da s a u, con $x_0 = s$ e $x_k = u$; p non ha cicli e, inoltre, il numero di nodi di p, ossia $k+1$, è inferiore o uguale alla cardinalità di V. Consideriamo il predecessore di u nel cammino e chiamiamolo v: se p è cammino minimo e $d[v] = \delta(s,v)$, dopo `relax(`v, u, w`)` vale $d[u] = \delta(s,u)$.

   Proprietà 12.1. Se esiste un cammino minimo da s a u con k archi, allora dopo il k-esimo ciclo dell'algoritmo (righe 2-4) vale $d[u] = \delta(s,u)$ (*invariante del ciclo*).

   Dimostrazione. Procediamo per induzione su k.

   Caso base: abbiamo $k = 0$, dunque $u = s$; si ha $d[s] = 0$ per `initSingleSource` e $\delta(s,s) = 0$ per l'ipotesi dell'assenza di cicli negativi, dunque la proprietà è dimostrata per il caso base.

   Ipotesi induttiva: supponiamo che la proprietà valga per $i < k$.

   Passo induttivo: sia $k > 0$; consideriamo il cammino $p = \langle x_0, ..., x_k \rangle$, con $x_0 = s$ e $x_k = u$. Per ipotesi induttiva, alla $k-1$-esima iterazione, $d[x_{k-1}] = \delta(s, x_{k-1})$; all'iterazione k-esima, tra le varie chiamate, viene eseguita `relax(`x_{k-1}, x_k, w`)` e, per il lemma 12.5, si ha $d[x_k] = \delta(s, x_k)$, ossia $d[u] = \delta(s,u)$. □

Dunque, al termine del ciclo, tutti i cammini sono stati sistemati (se non ci sono cicli negativi) e hanno al più $n-1$ archi.

Punto 1b. Segue direttamente da `relax`.

Punto 1c. Dobbiamo dimostrare che $d[v] \leq d[u] + w(u,v), \forall (u,v) \in E$. Per il lemma 12.2, si ha $\delta(s,v) \leq \delta(s,u) + w(u,v)$; consideriamo $(u,v) \in E$:

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

L'ultima uguaglianza segue dal punto 1a.

Punto 2. Dal punto 1 sappiamo che, se non ci sono cicli negativi raggiungibili da s, allora l'algoritmo ritorna *true*. Proviamo ora il viceversa, ossia che, se l'algoritmo ritorna *true*, non ci sono cicli negativi raggiungibili da s.
Sia $C = \langle x_0, ..., x_k \rangle$ un ciclo raggiungibile da s; poiché, per ipotesi, l'algoritmo ritorna true, si ha:

$$d[v] \leq d[u] + w(u,v)$$

In particolare, considerando gli archi (x_i, x_{i+1}) con $i = 0..k-1$:

$$\begin{aligned} & d[x_{i+1}] \leq d[x_i] + w(x_i, x_{i+1}) \\ & \sum_{i=0}^{k-1} d[x_{i+1}] \leq \sum_{i=0}^{k-1} d[x_i] + \sum_{i=0}^{k-1} w(x_i, x_{i+1}) \\ & d[x_1] + d[x_2] + ... + d[x_k] \leq d[x_0] + d[x_1] + ... + d[x_{k-1}] + \sum_{i=0}^{k-1} w(x_i, x_{i+1}) \\ & d[x_k] \leq d[x_0] + \sum_{i=0}^{k-1} w(x_i, x_{i+1}) \end{aligned}$$

Poiché C è un ciclo, si ha $x_0 = x_k$, ossia $d[x_0] = d[x_k]$; segue:

$$0 \leq \sum_{i=0}^{k-1} w(x_i, x_{i+1}) = w(C)$$

Abbiamo così dimostrato che, se l'algoritmo ritorna *true*, non esistono cicli negativi raggiungibili da s; segue dunque che, se ci sono cicli negativi raggiungibili da s, l'algoritmo restituisce *false*. □

Complessità

1. $O(n)$

2-4. vengono eseguite $m \cdot n$ chiamate a `relax`, il cui costo è $O(1)$; in totale si ha $O(m \cdot n)$

5-7. il ciclo viene eseguito m volte ed ogni iterazione ha costo $O(1)$; in totale si paga è $O(m)$

8. $O(1)$

Totale: $O(n) + O(m \cdot n) + O(m) + O(1) = O(m \cdot n)$.

L'algoritmo risulta essere computazionalmente più costoso rispetto a quello di Dijkstra, ma può essere applicato anche se gli archi hanno pesi negativi: la scelta di quale algoritmo utilizzare, dunque, dipende da come è definita la funzione peso.

Esempio di esecuzione

L'ordine di analisi degli archi segue l'ordine alfabetico con cui sono identificati i vertici: prima tutti gli archi che partono da a, poi quelli che partono da b, etc. Le figure si riferiscono a:

1. situazione dopo `initSingleSource`;
2. situazione dopo la prima passata di tutti gli archi;
3. situazione dopo la seconda passata di tutti gli archi (le successive non producono variazioni).

Chiaramente l'algoritmo restituisce *true*.

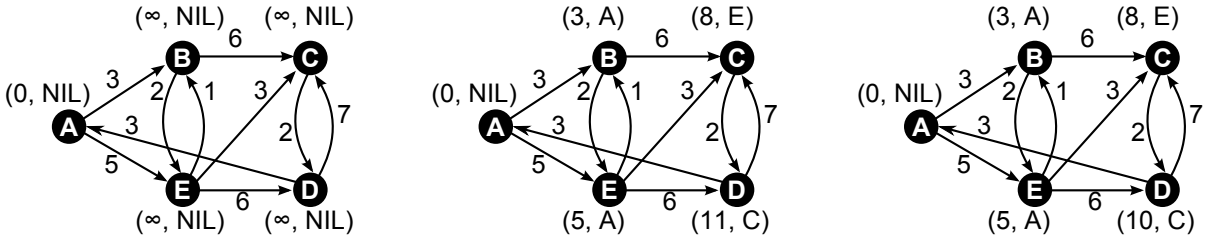


Figura 12.2: Esempio di esecuzione dell'algoritmo di Bellman - Ford (nodo sorgente a)

Capitolo 13

Cammini minimi fra tutte le coppie

Consideriamo il problema dei cammini minimi fra tutte le coppie in un grafo $G = (V, E, w)$ orientato, pesato, dove possono essere presenti archi (ma non cicli) di peso negativo; assumiamo, inoltre, che gli n vertici siano identificati da numeri interi da 1 a n.

13.1 Cammini minimi e moltiplicazione di matrici

Matrice di adiacenza pesata

Si tratta di una matrice $n \times n$ che, oltre a specificare se esiste l'arco (i, j), indica anche il peso di tale arco.

$$w_{i,j} = \begin{cases} 0 & \text{se } i = j \\ w(i,j) & \text{se } i \neq j \wedge (i,j) \in E \\ +\infty & \text{se } i \neq j \wedge (i,j) \notin E \end{cases}$$

Matrice delle distanze

è una matrice $n \times n$, aggiornata dall'algoritmo, dove $d_{i,j}$ contiene la stima della distanza tra i e j; al termine dell'esecuzione dell'algoritmo, vogliamo che $d_{i,j} = \delta(i,j), \forall i, j \in V$.

Moltiplicazione di matrici

Siano $i, j \in V$; definiamo:

$$\mathcal{C}_{i,j} = \{p = \langle x_0, ..., x_q \rangle : x_0 = i \wedge x_q = j \wedge (x_{l-1}, x_l) \in E, l = 1..q\}$$

La distanza tra i e j può essere definita come segue:

$$\delta(i,j) = \min_{p \in \mathcal{C}(i,j)} w(p)$$

Fissiamo $m \geq 1$:

$$\mathcal{C}_{i,j}^{(m)} = \{p0 \langle x_0, ..., x_q \rangle : x_0 = i \wedge x_q = j \wedge (x_{l-1}, x_l) \in E, l = 1..q \wedge q \leq m\}$$

$$d_{i,j}^{(m)} = \min_{p \in \mathcal{C}_{i,j}^{(m)}} w(p)$$

Sono valide le seguenti relazioni:

$$\mathcal{C}_{i,j}^{(1)} \subseteq \mathcal{C}_{i,j}^{(2)} \subseteq ... \subseteq \mathcal{C}_{i,j}^{(m)} \subseteq \mathcal{C}_{i,j}^{(m+1)} \subseteq ... \subseteq \mathcal{C}_{i,j}$$

$$\mathcal{C}_{i,j} = \bigcup_{m \in \mathbb{N}} \mathcal{C}_{i,j}^{(m)}$$

$$d_{i,j}^{(1)} \geq d_{i,j}^{(2)} \geq ... \geq d_{i,j}^{(m)} \geq ... \geq \delta(i,j)$$

Poiché non sono presenti, per ipotesi, cicli negativi, possiamo assumere che i cammini minimi siano cammini semplici, dunque formati al più da $n-1$ archi; ossia, si ha $d_{i,j}^{(n-1)} = d_{i,j}^{(n)} = d_{i,j}^{(n+1)} = ... = \delta(i,j)$. L'idea è di progettare un algoritmo per il calcolo di $d_{i,j}^{(m)}$ in modo che, quando arriviamo a $d_{i,j}^{(n-1)}$, otteniamo proprio le distanze; cerchiamo una formula ricorsiva per il calcolo di $d_{i,j}^{(m)}$, per induzione.

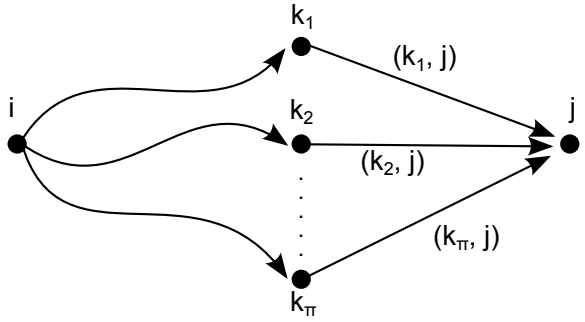
Caso base: si ha $m = 1$

$$\mathcal{C}_{i,j}^{(1)} = \begin{cases} \{\langle i \rangle\} & \text{se } i = j \\ \{\langle i,j \rangle\} & \text{se } i \neq j \wedge (i,j) \in E \\ \emptyset & \text{se } i \neq j \wedge (i,j) \notin E \end{cases}$$

$$d_{i,j}^{(1)} = \begin{cases} 0 & \text{se } i = j \\ w(i,j) & \text{se } i \neq j \wedge (i,j) \in E \\ +\infty & \text{se } i \neq j \wedge (i,j) \notin E \end{cases}$$

Ipotesi induttiva: supponiamo di essere riusciti a calcolare la matrice delle distanze $D^{(m-1)}$).

Passo induttivo: sia $m > 1$ e proviamo a calcolare $D^{(m)}$ usando l'ipotesi. Prendiamo $i, j \in V$: $p \in \mathcal{C}_{i,j}^{(m)}$ può essere scomposto come $p = p', j$, con $p' \in \mathcal{C}_{i,k}^{(m-1)}$; in generale, i cammini minimi in $\mathcal{C}_{i,j}^{(m)}$ sono scomponibili come segue:



da cui:

$$d_{i,j}^{(m)} = \min_{p \in \mathcal{C}_{i,j}^{(m)}} w(p) = \min_{\substack{k \in V \\ (k,j) \in E}} \left\{ d_{i,k}^{(m-1)} + w(k,j) \right\}$$

Sia $h \in V$: se $(h,j) \notin E$, allora $d_{i,h}^{(m-1)} + w(h,j) = +\infty$.

La formula cercata è la seguente:

$$d_{i,j}^{(m)} = \min_{k \in V} \left\{ d_{i,k}^{(m-1)} + w(k,j) \right\}$$

Per implementare la formula ricorsiva, definiamo la seguente funzione:

funzione `extendShortestPath` (*matrice delle distanze* $D^{(m-1)}$*, matrice di adiacenza pesata* W) $\rightarrow$ *matrice delle distanze* $D^{(m)}$

```
1 n = rows(D)
2 Dm : matrice[n][n]
3 for i = 1 to n do
4   for j = 1 to n do
5     Dm[i][j] = + ∞
6     for k = 1 to n do
7       Dm[i][j] = min(Dm[i][j], D[i][k] + W[k][j])
8 return Dm
```

Complessità: $\Theta(n^3)$.

La funzione definisce un nuovo tipo di prodotto tra matrici $D \otimes W = D'$, dove D' è la matrice calcolata da `extendShortestPath`:

algoritmo `showAllPairsShortestPath` (*matrice di adiacenza pesata* W) $\rightarrow$ *matrice delle distanze* $D^{(n-1)}$

```
1 n = rows(W)
2 D(1) = W
3 for m = 2 to n - 1 do
4   D(m) = D(m - 1) ⊗ W
5 return D(n - 1)
```

Complessità: $\Theta(n^4)$.

13.2 Algoritmo di Floyd-Warshall

Consideriamo un grafo che goda delle proprietà descritte ad inizio capitolo; vogliamo determinare $\delta(i, j)$ per ogni coppia i, j. Costruiamo $\mathcal{C}_{i,j}$ in maniera incrementale; sia $0 \leq k \leq n$ e definiamo:

$$P_{i,j}^{(k)} = \{p = \langle x_0, ..., x_q \rangle : x_0 = i \wedge x_q = j \wedge (x_{l-1}, x_l) \in E, l = 1..n \wedge x_h \leq k, h = 1..(q-1)\}$$

Si tratta dell'*insieme dei cammini* da i a j i cui *nodi interni* sono in $1..k$; osserviamo che $P_{i,j}^{(n)} = \mathcal{C}_{i,j}$ quindi, se definiamo:

$$d_{i,j}^{(k)} = \min_{p \in P_{i,j}^{(k)}} w(p)$$

si ha:

$$d_{i,j}^{(n)} = \delta(i, j)$$

L'obiettivo è calcolare $d_{i,j}^{(n)}$ per ogni coppia di vertici in V; procediamo per induzione su k.

Caso base: si ha $k = 0$; $P_{i,j}^{(0)}$ è l'insieme dei cammini da i a j che *non* hanno nodi interni, dunque:

$$P_{i,j}^{(0)} = \begin{cases} \{\langle i \rangle\} & \text{se } i = j \\ \{\langle i, j \rangle\} & \text{se } i \neq j \wedge (i, j) \in E \\ \emptyset & \text{se } i \neq j \wedge (i, j) \notin E \end{cases}$$

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{se } i = j \\ w(i, j) & \text{se } i \neq j \wedge (i, j) \in E \\ +\infty & \text{se } i \neq j \wedge (i, j) \notin E \end{cases}$$

Se chiamiamo $D^{(k)} = (d_{i,j}^{(k)})$, si ha $D^{(0)} = W$.

Ipotesi induttiva: supponiamo di conoscere $P_{i,j}^{(k-1)}$.

Passo induttivo: sia $k \geq 1$; usando l'ipotesi induttiva, cerchiamo di ottenere $P_{i,j}^{(k)}$. Innanzitutto, osserviamo che $P_{i,j}^{(k-1)} \subseteq P_{i,j}^{(k)}$ e definiamo $\widehat{P}_{i,j^{(k)}} = P_{i,j}^{(k)} - P_{i,j}^{(k-1)} = \left\{p \in P_{i,j}^{(k)} : p \text{ passa per il vertice } k\right\}$. Si ha:

$$\begin{aligned} d_{i,j}^{(k)} &= \min_{p \in P_{i,j}^{(k)}} w(p) \\ &= \min\left(\min_{p \in P_{i,j}^{(k-1)}} w(p), \min_{p \in \widehat{P}_{i,j}^{(k)}} w(p)\right) \end{aligned}$$

Il primo parametro della funzione *minimo* è:

$$\min_{p \in P_{i,j}^{(k-1)}} w(p) = d_{i,j}^{(k-1)}$$

Per quanto riguarda il secondo, dobbiamo trovare il peso del cammino più leggero da i a j che può passare per i nodi $1..k$ e passa sicuramente per k; consideriamo cammini semplici, in quanto non sono presenti cicli negativi: il cammino che cerchiamo può essere scomposto in due parti:

1. cammino da i a k che può passare per i nodi $1..k-1$;
2. cammino da k a j che può passare per i nodi $1..k-1$.

Chiaramente le due parti dovranno essere entrambe minime, affinché il cammino totale lo sia: il loro costo lo conosciamo ed è, rispettivamente, $d_{i,k}^{(k-1)}$ e $d_{k,j}^{(k-1)}$. Dunque, il secondo parametro della funzione *minimo* è:

$$\min_{p \in \widehat{P}_{1,j}^{(k)}} w(p) = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$$

La formula ricorsiva per il calcolo di $d_{i,j}^{(k)}$ è dunque la seguente:

$$d_{i,j}^{(k)} = \begin{cases} w(i,j) & \text{se } k = 0 \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}) & \text{se } k > 0 \end{cases}$$

L'algoritmo segue direttamente dalla formula:

algoritmo `floydWarshallCubico` (*matrice di adiacenza pesata* W) $\rightarrow$ *matrice delle distanze* $D^{(n)}$

```
1 n = rows(W)
2 for k = 1 to n do
3   for i = 1 to n do
4     for j = 1 to n do
5       d_{i,j}^(k) = min(d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1))
6 return D^(n)
```

Complessità

Temporale: $\Theta(n^3)$
Spaziale: $\Theta(n^3)$

Costruzione di un cammino minimo

Un metodo per costruire i cammini minimi usando l'algoritmo di Floyd Warshall consiste nell'usare la matrice dei pesi di cammino minimo prodotta e costruire, in tempo $O(n^3)$, la *matrice dei predecessori* Π. Un secondo metodo prevede di calcolare Π 'in linea' con l'algoritmo, o meglio, si calcola una sequenza di matrici $\Pi^{(0)}$, $\Pi^{(1)}$, ..., $\Pi^{(n)}$, dove $\Pi = \Pi^{(n)}$ e $\pi_{i,j}^{(k)}$ è definito come il predecessore di j su un cammino minimo dal vertice i avente tutti i vertici intermedi nell'insieme $\{1, ..., k\}$.
Si può definire induttivamente $\pi_{i,j}^{(k)}$; quando $k = 0$ si ha:

$$\pi_{i,j}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ o } w(i,j) = \infty \\ i & \text{se } i \neq j \text{ e } w(i,j) < \infty \end{cases}$$

Per $k > 1$, se prendiamo il cammino da i a j passante per k, allora il predecessore di j è lo stesso vertice che avevamo scelto come predecessore di j in un cammino da k con tutti i vertici intermedi nell'insieme $\{1, ..., k-1\}$; altrimenti, scegliamo lo stesso predecessore di j che avevamo scelto su un cammino minimo da i con tutti i vertici nell'insieme $\{1, ..., k-1\}$. Formalmente:

$$\pi_{i,j}^{(k)} = \begin{cases} \pi_{i,j}^{(k-1)} & \text{se } d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \\ \pi_{k,j}^{(k-1)} & \text{se } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \end{cases}$$

Miglioramento

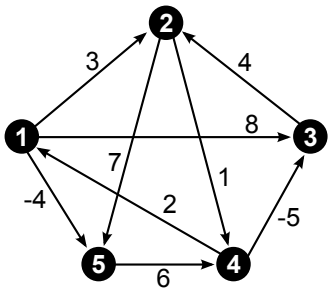
L'algoritmo che segue è una versione migliorata di Floyd Warshall ed ha complessità spaziale $\Theta(n^2)$, mentre la complessità temporale non cambia ($\Theta(n^3)$): l'idea consiste nell'utilizzare sempre la stessa matrice, invece di utilizzarne una diversa per ciascun k.

algoritmo `floydWarshallQuadratico` (*matrice di adiacenza pesata* W) $\rightarrow$ *matrice delle distanze* $D^{(n)}$

```
1 n = rows(W)
2 for k = 1 to n do
3   for i = 1 to n do
4     for j = 1 to n do
5       d_{i,j} = min(d_{i,j}, d_{i,k} + d_{k,j})
6 return D
```

Esempio di esecuzione

Simuliamo l'esecuzione sul seguente grafo:



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Capitolo 14

Teoria della NP-completezza

14.1 Complessità di problemi decisionali

In generale, possiamo pensare ad un problema P come a una relazione $P \subseteq i \times S$, dove I è l'insieme delle *istanze di ingresso* e S quello delle *soluzioni*; possiamo inoltre immaginare di avere un predicato che, presa un'istanza $x \in I$ ed una soluzione $s \in S$, restituisca 1 se $(x, s) \in P$ (ovvero s è soluzione di P sull'istanza s), 0 altrimenti. Usando questa terminologia, possiamo definire una prima classificazione dei problemi:

Problemi di decisione: sono problemi che richiedono una risposta binaria, dunque $S = \{0, 1\}$; in particolare, richiedono di verificare se l'istanza x soddisfa una certa proprietà.

Problemi di ricerca: data un'istanza x, questi problemi chiedono di restituire una soluzione s tale che $(x, s) \in P$.

Problemi di ottimizzazione: data un'istanza x, si vuole trovare la migliore soluzione s^* tra tutte le possibili soluzioni s per cui $(x, s) \in P$; la bontà della soluzione è valutata secondo un criterio specificato dal problema stesso.

I principali concetti della teoria della complessità computazionale sono stati definiti in termini dei problemi di decisione: si osservi, comunque, che è possibile esprimere problemi di altre categorie in forma decisionale (e la difficoltà della forma decisionale è al più pari a quella del problema iniziale), quindi caratterizzare la complessità di quest'ultimo permette di dare almeno una limitazione inferiore alla complessità del primo.

è necessario fissare un po' di notazione; dopo aver stabilito una codifica binaria *efficiente* dell'input, definiamo un problema decisionale P come una funzione, nel seguente modo:

$$P : \{0, 1\}^* \to \{0, 1\}$$

Inoltre, dato un problema decisionale P, definiamo il suo *linguaggio* come:

$$L = \left\{x \in \{0, 1\}^* : P(x) = 1\right\}$$

14.1.1 Classi di complessità

Dati un problema di decisione P ed un algoritmo A, diciamo che A risolve P se A restituisce 1 su un'istanza x se e solo se $(x, 1) \in P$. Inoltre, diciamo che A risolve P in tempo $t(n)$ e spazio $s(n)$ se il tempo di esecuzione e l'occupazione di memoria di A sono, rispettivamente, $t(n)$ e $s(n)$.

Definizione 14.1. Data una qualunque funzione $f(n)$, chiamiamo $\textsc{Time}(f(n))$ e $\textsc{Space}(f(n))$ gli insiemi dei problemi decisionali che possono essere risolti, rispettivamente, in tempo e spazio $O(f(n))$.

Definizione 14.2. La classe P è la classe dei problemi risolvibili in tempo polinomiale nella dimensione n dell'istanza di ingresso:

$$\mathrm{P} = \bigcup_{c=0}^{\infty} \textsc{Time}(n^c)$$

La classe PSpace è la classe dei problemi risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso:

$$\mathrm{PSPACE} = \bigcup_{c=0}^{\infty} \textsc{Space}(n^c)$$

La classe ExpTime è la classe dei problemi risolvibili in tempo esponenziale nella dimensione n dell'istanza di ingresso:

$$\textsc{ExpTime} = \bigcup_{c=0}^{\infty} \textsc{Time}(2^{n^c})$$

Si hanno le seguenti relazioni tra le classi: $\mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \textsc{ExpTime}$; non è noto se le inclusioni siano proprie o meno, in quanto sono quesiti di teoria della complessità ancora aperti.
Inoltre, la classe P è considerata come una classe soglia tra problemi trattabili e intrattabili.

14.2 La classe NP

Spesso, in caso di risposta affermativa ad un problema di decisione, si richiede anche di fornire un qualche oggetto y, dipendente dall'istanza x e dal problema specifico, che possa certificare il fatto che x soddisfi la proprietà richiesta: tale oggetto è noto come *certificato*. Un certificato è definito come una *stringa binaria* y ed una *funzione verificatrice*

$$g : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$$

tale che $x \in L$ se e solo se esiste y tale per cui $g(x,y) = 1$.
Se g lavora in tempo polinomiale e y è di lunghezza polinomiale rispetto alla lunghezza di x, allora L è verificabile in tempo polinomiale e per questo definiamo la classe di complessità NP: informalmente, essa è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale.

14.2.1 Non determinismo

Negli algoritmi visti finora, il passo successivo è sempre univocamente determinato dallo stato della computazione: per tale motivo, sono detti *deterministici*; un algoritmo non deterministico, invece, oltre alle normali istruzioni, può eseguirne alcune del tipo *indovina* $z \in \{0,1\}$, ovvero può 'indovinare' un valore binario per z facendo proseguire la computazione in una 'giusta' direzione.

Definizione 14.3. Data una qualunque funzione $f(n)$, chiamiamo $\mathrm{NTIME}(f(n))$ l'insieme dei problemi decisionali che possono essere risolti da un algoritmo non deterministico in tempo $O(f(n))$. La classe NP è la classe dei problemi risolvibili in tempo polinomiale non deterministico nella dimensione n dell'istanza di ingresso:

$$\mathrm{NP} = \bigcup_{c=0}^{\infty} \mathrm{NTIME}(n^c)$$

Un algoritmo non deterministico può essere suddiviso in due fasi:

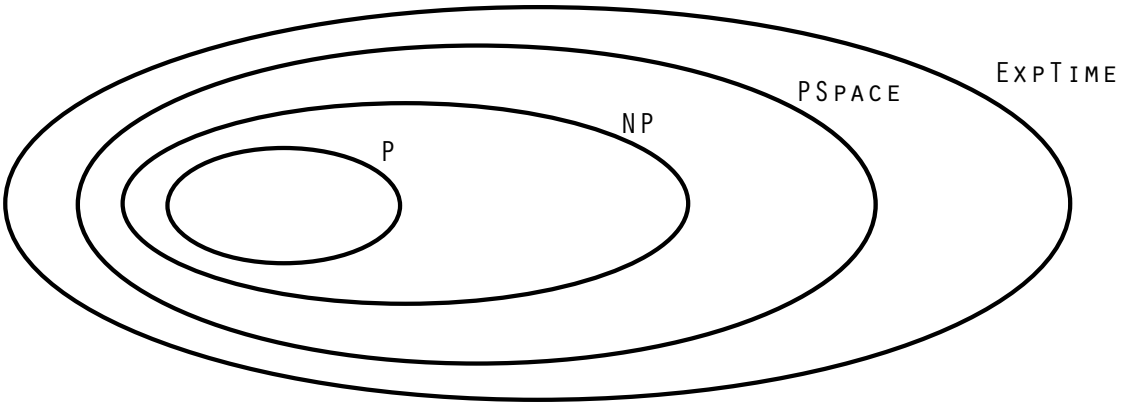
Fase non deterministica di costruzione: sfruttando la funzione *indovina*, l'algoritmo costruisce un certificato per l'istanza del problema.

Fase deterministica di verifica: l'algoritmo verifica che il certificato prodotto sia effettivamente una soluzione del problema per l'istanza data.

14.2.2 La gerarchia

è facile osservare che P $\subseteq$ NP, poiché un algoritmo deterministico è un caso particolare di uno non deterministico, in cui la funzione *indovina* non viene mai utilizzata. Inoltre, la fase deterministica di verifica può essere condotta in tempo polinomiale solo se il certificato ha dimensione polinomiale, da cui NP $\subseteq$ PSpace; si congettura, ma nessuno è ancora riuscito a dimostrarlo, che entrambe le inclusioni siano proprie, ovvero:

$$\text{P} \subset \text{NP} \subset \text{PSpace} \subset \text{ExpTime}$$



Si osservi che la gerarchia abbozzata, in realtà, è ben più complessa e vasta: infatti, esistono anche problemi *indecidibili*, ovvero non risolubili indipendentemente dalla quantità di tempo e memoria a disposizione, come il *problema della fermata*.

14.3 Riducibilità polinomiale

Un linguaggio L_1 è riducibile in tempo polinomiale a L_2 se esiste una funzione calcolabile in tempo polinomiale

$$r : \{0,1\}^* \to \{0,1\}^*$$

tale che, per ogni $x \in \{0,1\}^*$, $x \in L_1$ se e solo se $r(x) \in L_2$. r è detta *funzione di riduzione* e scriviamo $L_1 \leq_p L_2$.

Lemma 14.1. *Siano $L_1, L_2 \subset \{0,1\}^*$ linguaggi tali che $L_1 \leq_p L_2$: allora $L_2 \in P$ implica $L_1 \in P$.*

Dimostrazione. Visto che la funzione di riduzione lavora in tempo polinomiale, $r(x)$ ha lunghezza polinomiale rispetto alla lunghezza di x; ma un polinomio di un polinomio è ancora un polinomio, per cui $r(x) \in L_2$ viene deciso in tempo polinomiale rispetto alla lunghezza di x. □

14.4 Problemi NP-completi

Definizione 14.4. Un problema si dice NP-completo se:

- $L \in$ NP;
- $L' \leq_p L$ per ogni $L' \in$ NP.

Se vale solo la seconda condizione, L si dice essere *NP-difficile*; la classe dei problemi NP-completi si chiama NPC.

Consideriamo i seguenti problemi:

Satisfiability: una formula booleana in forma congiuntiva consiste in una serie di clausole congiunte, in cui ogni clausola è una serie di disgiunzioni di variabili, possibilmente negate; se ogni clausola contiene meno di k variabili si dice $k - CNF$. Il problema è trovare un assegnamento booleano che renda vera la formula: definiamo 3-SAT = $\{x : x$ è una formula in forma 3-CNF che ha almeno un assegna mento che la rende vera$\}$.

Clique: è un sottinsieme di vertici di un grafo per cui ognuno è connesso ad ogni altro del sottinsieme. Ovviamente le coppie di vertici connesse for- mano delle clique, ma potrebbero esserci insiemi più grandi a cui siamo interessati. Il linguaggio corrispondente è CLIQUE = $\{\langle G, k\rangle : G$ è un grafo con almeno una clique di dimensione $k\}$.

Independent set: è un sottinsieme di vertici che non sono collegati l'uno con l'altro. Definiamo IN-DEPENDENT SET = $\{\langle G, k\rangle$: G è un grafo con almeno un independent set di dimensione $k\}$.

Assumiamo che 3-SAT sia NP-completo e vediamo come si può ridurre il problema in tempo polinomiale a CLIQUE, e quest'ultimo ridurlo in tempo polinomiale a INDIPENDENT SET.

Prima parte: 3-SAT $\leq_p$ CLIQUE.

Presa una formula in forma 3-CNF con m clausole, costruiamo un grafo G che abbia un vertice per ogni variabile (o variabile negata) all'interno di ogni clausola, quindi un grafo con $3 \cdot m$ vertici. Colleghiamo tra loro due vertici se e solo se appartengono a clausole diverse e non rappresentano la stessa variabile negata, ossia non sono una coppia della forma $(x, \overline{x})$. La funzione $r(\phi)$ restituisce una coppia $\langle G, m\rangle$ lavora in tempo polinomiale, e ci chiediamo se $\phi \in$ 3-SAT se e solo se $\langle G, m\rangle \in$ CLIQUE.
Se G contenesse una clique di dimensione m, questa avrebbe un vertice in ogni clausola e ci consentirebbe di trovare un'assegnazione che soddisfa ϕ senza incappare in una contraddizione. Viceversa se avessimo un'assegnazione che soddisfa ϕ, potremmo scegliere da ogni clausola una variabile con valore 'vero', dal momento che le m variabili scelte hanno tutte lo stesso valore, quindi dovrebbero essere collegate l'una con l'altra nel grafo, formando una clique.

Seconda parte: CLIQUE $\leq_p$ INDEPENDENT SET.

Dato un grafo G, consideriamo il suo complementare $\overline{G}$: è sufficiente osservare che un sottoinsieme di vertici è una clique in G se e solo se è un independent set in $\overline{G}$.

14.5 La classe coNP e la relazione tra P e NP

Definizione 14.5. Il complemento di un linguaggio $L \subset \{0,1\}^*$ è l'insieme delle stringhe $x \notin L$.

Definizione 14.6. La classe coNP consiste di linguaggi il cui complemento appartiene a NP:

$$coNP = \left\{L : \overline{L} \in NP\right\}$$

Si ha $P \subseteq coNP$ e $P = coP$, da cui $P \subseteq NP \cap coNP$; questo, a sua volta, implica $P = NP \Rightarrow coP = coNP \Rightarrow NP = coNP$. Quindi $NP \neq coNP \Rightarrow P \neq NP$. Ci sono quattro possibilità:

- $P = NP = coNP$;
- $P \subset NP = coNP$;
- $P = coNP \cap NP, coNP \cap NP \neq NP, NP \neq coNP$;
- $P \subset coNP \cap NP, coNP \cap NP \neq NP, NP \neq coNP$.

Nonostante la ricerca sia ancora aperta, generalmente si ritiene che $P \neq NP$, in quanto si ritiene strano che *cercare* una risposta sia tanto facile quanto *riconoscerla*.

Capitolo 15

Appendice

15.1 Serie aritmetica

Lemma 15.1. *La somma dei primi n numeri consecutivi ha il seguente valore:*

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

Dimostrazione. Definiamo

$$S_n = \sum_{i=1}^{n} i$$

Possiamo scrivere:

$$S_n = 1 + 2 + ... + n$$
$$S_n = n + (n-1) + ... + 1$$

Sommando membro a membro, si ottiene $2 \cdot S_n = n \cdot (n+1)$, da cui segue quanto si vuole dimostrare. □

15.2 Serie geometrica

Lemma 15.2. *La serie geometrica di ragione q, i cui addendi sono caratterizzati dall'avere una base costante $q \neq 1$ ed un esponente variabile, ha il seguente valore:*

$$\sum_{i=0}^{k} q^i = \frac{q^{k+1} - 1}{q - 1}$$

Dimostrazione. Sia

$$S_k(q) = \sum_{i=0}^{k} q^i$$

Moltiplicando ambo i membri per q, si ha:

$$q \cdot S_k(q) = \sum_{i=1}^{k+1} q^i$$

Sottraendo $S_k(q)$ ad entrambi i membri, si ottiene:

$$(q-1) \cdot S_k(q) = \sum_{i=1}^{k+1} q^i - \sum_{i=0}^{k} q^i = q^{k+1} - 1$$

da cui

$$S_k(q) = \frac{q^{k+1} - 1}{q - 1}$$

come volevamo dimostrare. □

15.3 Calcolo di somme per integrazione

Sono valide le seguenti disuguaglianze:

- se $f(x)$ è una funzione non decrescente:

$$\int_{a-1}^{b} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)dx$$

- se $f(x)$ è una funzione non crescente:

$$\int_{a}^{b+1} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)dx$$