Algoritmi randomizzati.

Livio Colussi

25 maggio 2016

1 Introduzione

Abbiamo già visto alcuni esempi di randomizzazione nella progettazione di algoritmi, quali ad esempio: l'algoritmo QUICK-SORT randomizzato per l'ordinamento di un array, l'algoritmo RAND-SELECT per la ricerca del k-esimo elemento in ordine di grandezza in un array, l'algoritmo dell'impronta di Rabin e Karp per la ricerca di una parola in un testo, l'hashing universale, ecc.

L'uso della randomizzazione permette spesso di ottenere un algoritmo più semplice o un algoritmo più veloce o entrambe le cose.

La randomizzazione può essere usata in modi diversi e per diversi scopi. Ne elenchiamo alcuni in modo informale.

Confondere l'avversario. Si usa quando la progettazione di un algoritmo può essere vista come un gioco tra il progettista dell'algoritmo ed un avversario.

L'avversario ha un qualche possibile guadagno da massimizzare (il tempo calcolo dell'algoritmo, la quantità di memoria usata, il costo della soluzione prodotta dall'algoritmo, ecc.) ed il progettista dell'algoritmo cerca di minimizzare il guadagno dell'avversario.

L'avversario conosce la strategia dell'algoritmo ed è in grado quindi di scegliere un input particolarmente cattivo per l'algoritmo.

Il guadagno dell'avversario su tale input è dunque un limite inferiore per la complessità dell'algoritmo nel caso pessimo.

Ad esempio se l'algoritmo è QUICK-SORT l'avversario può scegliere sempre un input già ordinato in modo che il tempo calcolo risulti $\Omega(n^2)$.

In questo caso la randomizzazione aiuta a confondere l'avversario impedendogli di poter prevedere le mosse dell'algoritmo.

Se l'algoritmo è randomizzato l'avversario conosce la strategia dell'algoritmo ma non conosce le scelte casuali dovute alla randomizzazione.

La cosa migliore che può fare l'avversario in questo caso è scegliere un input che massimizza il valore medio del suo guadagno rispetto alle possibili scelte casuali dell'algoritmo. Tale valore medio si assume quindi come complessità dell'algoritmo randomizzato.

Nel caso di QUICK-SORT randomizzato abbiamo dimostrato che tale valore è $\Theta(n \log n)$, pari alla complessità del miglior algoritmo di ordinamento deterministico.

Come vedremo in seguito ci sono problemi per i quali la complessità media dell'algoritmo randomizzato risulta strettamente minore della complessità (nel caso pessimo) di ogni algoritmo deterministico.

Campionamento casuale. L'idea che un campione casuale tratto da una popolazione sia rappresentativo dell'intera popolazione è alla base di tutte le indagini statistiche.

Questa idea viene spesso usata nella realizzazione di algoritmi randomizzati. Un esempio è il calcolo di volumi: il volume da valutare viene incluso in un volume più grande di valore noto, si generano dei punti casuali all'interno del volume grande e si contano quanti di essi cadono all'interno del volume da valutare. Ci si aspetta che il rapporto tra il numero di punti interni al volume da valutare e il numero totale di punti generati sia ragionevolmente vicino al rapporto tra i due volumi.

Su questa idea si basa anche quello che forse è il più vecchio algoritmo randomizzato conosciuto: l'algoritmo degli aghi di Buffon (1777) per calcolare il valore di π: Su di un foglio di carta su cui sono tracciate delle rette parallele a distanza d vengono lanciati degli aghi di lunghezza $\ell < d$ e si contano quanti di tali aghi vanno ad intersecare una delle linee.

La probabilità di una tale intersezione si può calcolare analiticamente e vale $\frac{2\ell}{d\pi}$ e quindi se lanciando n aghi se ne contano m incidenti le linee parallele possiamo concludere che $\pi \approx \frac{2\ell m}{dn}$.

Abbondanza di testimoni. Talvolta il problema da risolvere è del tipo "L'input gode di una certa proprietà P?". Ad esempio "È vero che il numero n è un numero composto?" (ossia non è un numero primo).

Generalmente la proprietà si può verificare fornendo un "testimonio" (nell'esempio un divisore di n). Spesso è però difficile trovare un testimonio con un algoritmo deterministico. In questi casi può essere di aiuto la randomizzazione.

Se riusciamo a trovare uno spazio di probabilità in cui i testimoni sono molto abbondanti, ad un algoritmo randomizzato possono bastare poche estrazioni casuali per riuscire a pescare un testimonio.

Se, ripetendo le estrazioni casuali, si riesce a trovare un testimonio abbiamo la dimostrazione che l'input soddisfa la proprietà. In caso contrario abbiamo un segnale forte (ma non la sicurezza) che l'input non soddisfi la proprietà.

Un algoritmo randomizzato che può ritornare una risposta sbagliata (sia pure con bassa probabilità) viene detto algoritmo di tipo *Monte Carlo*. Un algoritmo randomizzato che al contrario sia garantito ritornare la risposta giusta viene detto algoritmo di tipo *Las Vegas*.

Verifica di una identità. Ad esempio potremmo voler verificare se una funzione $f(x_1, \ldots, x_n)$ di n variabili è identicamente uguale a 0.

Un modo per risolvere questo problema consiste nel generare casualmente i valori $a_1, \ldots, a_n$ e calcolare $f(a_1, \ldots, a_n)$. Se si ottiene un valore diverso da 0 la risposta è chiaramente negativa.

Supponiamo di saper generare i valori casuali $a_1, \ldots, a_n$ con una distribuzione di probabilità tale che

$$\mathbf{Pr}[f(a_1, \ldots, a_n) = 0 \mid f(x_1, \ldots, x_n) \not\equiv 0] \leq c$$

con c costante minore di 1 (ad esempio $c = \frac{1}{2}$).

Possiamo allora determinare se $f(x_1, \ldots, x_n) \equiv 0$ con una probabilità arbitrariamente alta. Per fare questo basta ripetere un sufficiente numero di volte la verifica di $f(a_1, \ldots, a_n) = 0$ generando ogni volta un nuovo insieme di valori casuali $a_1, \ldots, a_n$.

Osserviamo che la verifica di una identità $f(x_1, \ldots, x_n) \equiv 0$ non è che un caso speciale del punto precedente in cui un testimonio è un insieme di valori $a_1, \ldots, a_n$ tale che $f(a_1, \ldots, a_n) \neq 0$.

Riordinamento casuale dell'input. L'efficienza di un algoritmo può dipendere dall'ordine in cui vengono presentati i dati in input; possiamo eliminare questa dipendenza usando la randomizzazione.

Un esempio classico è l'algoritmo di ordinamento QUICK-SORT che richiede tempo $O(n^2)$ nel caso pessimo ma mediante la randomizzazione il valore atteso del tempo calcolo si riduce a $O(n \log n)$ ed il tempo calcolo viene reso indipendente dall'ordine dei dati in input (dipende soltanto dalle scelte casuali dell'algoritmo).

Osserviamo che questo può essere visto come caso speciale del primo punto. In particolare QUICK-SORT randomizzato è di tipo Las Vegas; l'output è sempre correttamente ordinato.

Utilizzo di una impronta. Questa tecnica permette di rappresentare oggetti complessi mediante una piccola impronta. In particolari situazioni l'uguaglianza delle impronte di due oggetti costituisce un segnale forte del fatto che i due oggetti siano identici. Un esempio che abbiamo visto è l'algoritmo di pattern matching di Rabin e Karp.

Questa stessa idea è alla base della tecnica di *hashing* in cui un piccolo sottoinsieme S di un grande insieme U (l'insieme di tutte le possibili chiavi) viene mappato in un piccolo insieme M (le posizioni nella tavola hash) con la garanzia che elementi distinti di S abbiano un'alta probabilità di essere mappati in elementi distinti di M.

Utilizzo di catene di Markov. È utile per problemi di conteggio quali ad esempio contare il numero di cicli di un grafo o il numero di alberi o di matching o altro ancora.

Il problema viene dapprima trasformato in un problema di campionamento in uno spazio che generalmente ha dimensione esponenziale nella dimensione dell'input. Le catene di Markov permettono di generare casualmente dei punti in tale spazio in modo molto efficiente.

Perché i punti risultino effettivamente casuali occorre però che tali catene di Markov convergano rapidamente alla distribuzione uniforme. Catene di Markov di questo tipo vengono dette a rapida mescola.

Prima di illustrare con degli esempi tutti questi modi di utilizzare la tecnica della randomizzazione richiamiamo, nel prossimo paragrafo, alcuni concetti e risultati di calcolo delle probabilità che ci saranno utili nella progettazione e analisi degli algoritmi randomizzati (per approfondimenti vedi R. Karp [7]).

2 Alcune nozioni preliminari

Un algoritmo randomizzato è semplicemente un algoritmo che ha accesso ad un generatore di bit casuali. Il comportamento di tale algoritmo non è quindi determinato unicamente dall'input ma dipende anche dai bit casuali che vengono generati.

Possiamo immaginare che il generatore di bit casuali funzioni off-line generando una sequenza y, potenzialmente infinita, di bit casuali e che l'algoritmo vada a prendere i bit casuali da tale sequenza.

In questo caso, una volta generata la sequenza di bit casuali il comportamento dell'algoritmo risulta determinato unicamente dall'input e quindi esso può essere visto come un algoritmo deterministico.

Possiamo quindi pensare un algoritmo randomizzato come una famiglia di algoritmi deterministici, uno per ogni possibile sequenza potenzialmente infinita $y = b_1 b_2 \dots b_k \dots$ di bit casuali.

Scriveremo A_y per indicare l'algoritmo deterministico relativo alla sequenza di bit casuali $y \in \{0,1\}^\infty$ e indicheremo con $\mathcal{A} = \{A_y \mid y \in \{0,1\}^\infty\}$ la famiglia di algoritmi deterministici che costituisce l'algoritmo randomizzato.

Naturalmente l'algoritmo, dovendo terminare in un tempo finito, userà soltanto una parte iniziale finita $y_m = b_1 \dots b_m$ di tale sequenza la cui lunghezza m dipende soltanto dal particolare input x. In particolare se $T(x)$ è il tempo calcolo richiesto dall'algoritmo A_y per processare un input x il numero di bit della sequenza infinita y che vengono usati dall'algoritmo sarà limitato superiormente da $m(x) = O(T(x))$.

La complessità di un algoritmo randomizzato $\mathcal{A} = \{A_y\}$ si calcola come media delle complessità degli algoritmi deterministici che lo costituiscono.

Ad esempio consideriamo l'algoritmo randomizzato ITER riportato nel Programma 1.

```
ITER(n)
1  m = 0
2  while n ≠ 0
3       b = RANDOM()
4       n = b⌊n/2⌋
5       m = m + 1
6  return m
```

Programma 1: Il programma ITER.

Il numero m di iterazioni eseguite da tale algoritmo dipende sia dall'input n che dalla sequenza di bit casuali.

Consideriamo il caso in cui l'input sia $n = 5$. Con probabilità $\frac{1}{2}$ il primo bit generato è 0 nel qual caso l'algoritmo termina immediatamente in un solo passo: l'algoritmo deterministico A_y eseguito in questo caso è quindi uno relativo ad una sequenza $y = 0\ldots$ che inizia con il bit 0. Se la sequenza y è scelta casualmente questo succede con probabilità $\frac{1}{2}$.

La probabilità che il primo bit sia 1 è anch'essa $\frac{1}{2}$. In questo caso viene posto $n = 2$ e viene generato un secondo bit.

Se il secondo bit è 0 l'algoritmo termina in due passi: l'algoritmo deterministico A_y eseguito è uno relativo ad una sequenza $y = 10\ldots$ che inizia con i due bit 10 e la probabilità che un tale algoritmo venga scelto è $\frac{1}{4}$.

Se il secondo bit è 1 viene posto $n = 1$ e viene generato un terzo bit. Questa volta sia che il bit sia 0 sia che esso sia 1 viene posto $n = 0$ e l'algoritmo termina. Dunque per ogni sequenza di bit casuali $y = 11\ldots$ che inizia con i due bit 11 l'algoritmo termina in 3 passi. Se y è scelta casualmente questo succede con probabilità $\frac{1}{4}$.

Quindi il nostro algoritmo con input $n = 5$ richiede $m = 1$ passi con probabilità $\frac{1}{2}$, $m = 2$ passi con probabilità $\frac{1}{4}$ ed $m = 3$ passi con probabilità $\frac{1}{4}$. Èevidente che nel calcolare il numero medio di passi dobbiamo tener conto delle probabilità.

In altre parole dobbiamo calcolare quello che gli statistici chiamano il *valore atteso*, ossia:

$$\mathbf{E}[m] = 1\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{4} = \frac{7}{4}$$

Valore Atteso

In generale il valore atteso di una variabile casuale X (quale m nell'esempio) si indica con $\mathbf{E}[X]$ e si calcola come

$$\mathbf{E}[X] = \sum_{x \in X} x p_x$$

Una notevole proprietà del valore atteso è la linearità: date n variabili casuali $X_1, \ldots, X_n$ il valore atteso della loro somma è uguale alla somma dei valori attesi; in formula

$$\mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i]$$

Da notare che questa proprietà è vera anche se le variabili casuali non sono indipendenti.

L'analoga proprietà

$$\mathbf{E}\left[\Pi_{i=1}^{n} X_i\right] = \Pi_{i=1}^{n} \mathbf{E}[X_i]$$

per il prodotto di variabili casuali vale solo se le variabili sono indipendenti. In particolare per ogni costante c ed ogni variabile casuale X vale

$$\mathbf{E}[cX] = c\mathbf{E}[X]$$

in quanto la costante c vista come variabile casuale ha valore atteso $\mathbf{E}[c] = c$ ed è indipendente da qualsiasi altra variabile casuale.

Tornando all'esempio, possiamo calcolare il valore atteso del numero di passi $m(n)$ in funzione di n nel modo seguente:

$$\mathbf{E}[m(n)] = \frac{\lfloor \log n \rfloor + 1}{2^{\lfloor \log n \rfloor + 1}} + \sum_{i=1}^{\lfloor \log n \rfloor + 1} \frac{i}{2^i}$$

dove $\lfloor \log n \rfloor + 1$ è il numero di volte che occorre dividere n per 2 per arrivare a 0, il primo termine corrisponde alla sequenza di $\lfloor \log n \rfloor + 1$ bit casuali tutti uguali ad 1 mentre la sommatoria si riferisce ai casi in cui dopo $i - 1$ bit uguali ad 1 viene estratto un bit 0. Estraendo dalla sommatoria l'ultimo addendo e aggiungendolo al termine fuori della sommatoria che è uguale ad esso

$$\mathbf{E}[m(n)] = \frac{\lfloor \log n \rfloor + 1}{2^{\lfloor \log n \rfloor}} + \sum_{i=1}^{\lfloor \log n \rfloor} \frac{i}{2^i}$$

Infine, usando la formula:

$$\sum_{i=1}^{k} \frac{i}{2^i} = 2 - \frac{k+2}{2^k}$$

si ottiene

$$\mathbf{E}[m(n)] = \frac{\lfloor \log n \rfloor + 1}{2^{\lfloor \log n \rfloor}} + 2 - \frac{\lfloor \log n \rfloor + 2}{2^{\lfloor \log n \rfloor}} = 2 - \frac{1}{2^{\lfloor \log n \rfloor}}$$

Dunque, qualunque sia il valore di n vengono eseguite mediamente meno di 2 iterazioni.

Variabili indicatrici

Un altro strumento utile nell'analisi di algoritmi randomizzati sono le *variabili indicatrici*. Una variabile indicatrice può assumere soltanto uno dei due valori 0 o 1 e viene usata di solito quando si vuole rappresentare il fatto che un certo oggetto appartenga o meno ad un dato insieme. Illustriamo l'utilità delle variabili indicatrici con il seguente esempio.

Esempio 2.1 Una nave arriva in porto e i 40 marinai scendono tutti a terra. A tarda notte essi risalgono sulla nave ed essendo ubriachi fradici scelgono in modo casuale una branda libera in cui andare a dormire. Quale è il valore atteso del numero m di marinai che vanno a dormire nella propria branda?

Un approccio inefficiente è quello di calcolare il valore atteso sommando su tutte le $40!$ permutazioni possibili.

Un modo più efficiente si ottiene introducendo, per ogni marinaio una variabile indicatrice X_i il cui valore è 1 se l'i-esimo marinaio dorme nella sua branda e 0 altrimenti.

Il numero di marinai che dormono nella loro stessa branda è quindi $m = \sum_{i=1}^{40} X_i$. Siccome il valore atteso di X_i è

$$\mathbf{E}[X_i] = 1\frac{1}{40} + 0\frac{39}{40} = \frac{1}{40}$$

il valore atteso di m è

$$\mathbf{E}[m] = \mathbf{E}\left[\sum_{i=1}^{40} X_i\right] = \sum_{i=1}^{40} \mathbf{E}[X_i] = \sum_{i=1}^{40} \frac{1}{40} = 1$$

Osserviamo che nell'esempio le variabili X_i non sono indipendenti.

Esercizio 1 [J. von Neumann] Supponendo di avere a disposizione una moneta truccata della quale non sappiamo quale sia la probabilità p che il risultato sia testa. Come possiamo usare tale moneta per simulare il lancio di una moneta non truccata il cui risultato abbia le probabilità $\mathbf{Pr}[testa] = \mathbf{Pr}[croce] = 1/2$?

Descrivere un procedimento per cui il valore atteso del numero di lanci della moneta truccata necessari per simulare un lancio della moneta non truccata non sia maggiore di $\frac{1}{p(1-p)}$.

Esercizio 2 Supponendo di avere a disposizione una moneta non truccata trovare un procedimento per generare una permutazione casuale di n oggetti. Il procedimento deve essere efficiente sia in termini di tempo calcolo sia in termini di numero di lanci della monetina.

Ricorrenze probabilistiche

Talvolta la randomizzazione viene applicata ad un algoritmo ricorsivo lineare[1].

Ad esempio la struttura di un tale algoritmo potrebbe essere la seguente:

Algoritmo 1 *Data una istanza del problema di dimensione n viene fatta una scelta casuale in base alla quale il problema viene ridotto alla soluzione di una istanza del problema di dimensione minore. La scelta casuale e la conseguente riduzione viene quindi ripetuta finché non si arrivi ad una istanza del problema di dimensione minore o uguale ad 1 (che si risolve direttamente).*

Un esempio di algoritmo di questo tipo è l'algoritmo di selezione randomizzato RAND-SELECT che dato un array di n elementi non ordinato ed un intero k compreso tra 1 ed n trova il k-esimo elemento in ordine di grandezza. L'algoritmo RAND-SELECT scritto in pseudo codice è riportato nel Programma 2.

```
RAND-SELECT(A, p, r, k)  // 1 ≤ k ≤ r − p + 1
1  if p == r
2       return A[p]
3  q = RAND-PARTITION(A, p, r)
4  i = q − p + 1
5  if k == i   // Il pivot della partizione è il k-esimo elemento
6       return A[q]
7  elseif k < i
8       return RAND-SELECT(A, p, q − 1, k)
9  else return RAND-SELECT(A, q + 1, r, k − i)
```

Programma 2: Il programma RAND-SELECT.

L'algoritmo sceglie casualmente un elemento dell'array, effettua la partizione degli elementi dell'array mettendo all'inizio tutti gli elementi minori dell'elemento scelto e a destra tutti quelli maggiori. Procede quindi la ricerca nel segmento di array che contiene il k-esimo elemento. In questo modo il problema di dimensione n viene ridotto ad un sottoproblema di dimensione $n - X$ con X variabile casuale il cui valore è compreso tra 1 ed $n - 1$.

Dato un algoritmo di questo tipo supponiamo di sapere che per una istanza del problema di dimensione n la riduzione X è in media almeno $g(n)$, ossia $\mathbf{E}[X] \geq$

[1]Ricordiamo che un algoritmo ricorsivo si dice lineare quando effettua al più una sola chiamata ricorsiva ad ogni livello; l'albero delle chiamate ricorsive si riduce quindi ad una lista lineare.

$g(n)$. Siccome ci aspettiamo riduzioni maggiori per istanze del problema maggiori possiamo anche supporre che la funzione $g(n)$ sia non decrescente.

Ad esempio, per RAND-SELECT possiamo prendere $g(n) = n/4$. La dimostrazione che $\mathbf{E}[X] \geq n/4$ è la seguente.

Supponiamo che l'elemento scelto casualmente sia l'i-esimo. Se $i = k$ si escludono tutti gli altri elementi e quindi la riduzione è $X = n - 1$. Se $i < k$ si escludono tutti gli elementi di indice minore o uguale di i e la riduzione è $X = i$. Infine se $i > k$ si escludono tutti gli elementi di indice maggiore o uguale i e la riduzione è $X = n - i + 1$.

Siccome ogni valore di i ha probabilità $1/n$ di essere scelto

$$\begin{aligned}
\mathbf{E}[X] &= \frac{1}{n}(n-1) + \sum_{i=1}^{k-1} \frac{1}{n} i + \sum_{i=k+1}^{n} \frac{1}{n}(n-i+1) \\
&= \frac{1}{n}\left[n - 1 + \frac{k(k-1)}{2} + \frac{(n-k+1)(n-k)}{2}\right] \\
&= \frac{2k^2 - 2nk - 2k + n^2 + 3n - 2}{2n}
\end{aligned}$$

Derivando rispetto a k si ottiene

$$\frac{d\,\mathbf{E}[X]}{dk} = \frac{2k - n - 1}{n}$$

La derivata si annulla quando $k = (n+1)/2$. Siccome la derivata è sempre crescente, per $k = (n+1)/2$ la funzione ha un minimo e quindi

$$\begin{aligned}
\mathbf{E}[X] &\geq \frac{2[(n+1)/2]^2 - 2n[(n+1)/2] - 2[(n+1)/2] + n^2 + 3n - 2}{2n} \\
&= \frac{n}{4} + 1 - \frac{5}{4n} > n/4
\end{aligned}$$

dove l'ultima disuguaglianza vale in quanto la chiamata ricorsiva viene effettuata solo per $n > 1$.

Tornando al caso generale, supponiamo che per un generico algoritmo ricorsivo lineare randomizzato valga la limitazione $\mathbf{E}[X] \geq g(n)$ e sulla base soltanto di questo vogliamo stimare il numero $T(n)$ di chiamate ricorsive che vengono effettuate dall'algoritmo su di una istanza del problema di dimensione n.

Naturalmente $T(n)$ è una variabile casuale che dipende dalle scelte casuali dell'algoritmo. Ci interessa quindi il valore atteso $\mathbf{E}[T(n)]$.

Il seguente teorema fornisce un limite superiore per $\mathbf{E}[T(n)]$ in funzione del limite inferiore $g(n)$ per $\mathbf{E}[X]$.

Teorema 2.2 *Dato un algoritmo randomizzato lineare ricorsivo per il quale il valore atteso della diminuzione X nella dimensione dell'istanza del problema è limitato inferiormente da $\mathbf{E}[X] \geq g(n)$ con $g(n)$ non decrescente, il valore atteso del numero di chiamate ricorsive è limitato superiormente da*

$$\mathbf{E}[T(n)] \leq \sum_{i=2}^{n} \frac{1}{g(i)}$$

Dimostrazione. La dimostrazione è per induzione su n. Per $n = 1$ non vengono effettuate chiamate ricorsive per cui $\mathbf{E}[T(1)] = 0 = \sum_{i=2}^{1} \frac{1}{g(i)}$ banalmente. Supponiamo quindi vera la disuguaglianza per tutte le istanze di dimensione minore di n. Chiaramente $T(n) = 1 + T(n - X)$ e quindi

$$\begin{aligned}
\mathbf{E}[T(n)] &= 1 + \mathbf{E}[T(n - X)] \\
&\leq 1 + \mathbf{E}\left[\sum_{i=2}^{n-X} \frac{1}{g(i)}\right] \\
&= 1 + \mathbf{E}\left[\sum_{i=2}^{n} \frac{1}{g(i)}\right] - \mathbf{E}\left[\sum_{i=n-X+1}^{n} \frac{1}{g(i)}\right] \\
&\leq 1 + \sum_{i=2}^{n} \frac{1}{g(i)} - \mathbf{E}\left[\sum_{i=n-X+1}^{n} \frac{1}{g(n)}\right] \\
&= 1 + \sum_{i=2}^{n} \frac{1}{g(i)} - \frac{1}{g(n)}\mathbf{E}[X] \\
&\leq \sum_{i=2}^{n} \frac{1}{g(i)} \quad \square
\end{aligned}$$

Applicando il Teorema precedente all'algoritmo di selezione troviamo

$$\mathbf{E}[T(n)] \leq \sum_{i=2}^{n} \frac{4}{i} \leq 4(H_n - 1) \leq 4 \ln n = O(\log n)$$

dove $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$ è l'n-esimo numero armonico per il quale vale la diseguaglianza $H_n \leq \ln n + 1$.

Limiti di Chernoff

Abbiamo visto come la complessità di un algoritmo randomizzato si misuri con il valore atteso calcolato su tutte le possibili scelte casuali.

Il valore atteso ci dice quale sia la complessità che ci possiamo aspettare mediamente quando eseguiamo l'algoritmo randomizzato.

Per poter valutare un algoritmo randomizzato talvolta la complessità media può non essere sufficiente: due algoritmi randomizzati aventi la stessa complessità media possono in realtà comportarsi in modo molto diverso. Il fatto che in media ogni italiano mangi mezzo pollo può indicare due situazioni ben diverse: che ogni italiano mangia circa mezzo pollo chi un po' di più chi un po' di meno oppure che metà degli italiani mangiano un pollo intero mentre l'altra metà non mangia niente.

Più seriamente ci interessa sapere quanto sia probabile che ci si discosti molto dal valore atteso. Per questo tipo di valutazione si possono usare i limiti di Chernoff che forniscono una maggiorazione della probabilità che un valore casuale risulti maggiore di un dato multiplo del valore atteso o minore di un dato sottomultiplo.

Vi sono molti tipi di limiti di Chernoff applicabili a varie situazioni. Quelli che noi useremo, e che riportiamo senza dimostrazione, sono i seguenti.

Teorema 2.3 *Siano $X_1, X_2, \ldots, X_n$ variabili casuali binarie indipendenti tali che $\mathbf{Pr}[X_i = 1] = p_i$ con $p_i < 1$ e sia $X = \sum_{i=1}^{n} X_i$ la variabile casuale somma e $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} p_i$ il suo valore atteso. Allora, per ogni $\delta > 0$ abbiamo $\mathbf{Pr}[X > (1+\delta)\mu] < F^+(\mu, \delta)$ con*

$$F^+(\mu, \delta) = \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu$$

Teorema 2.4 *Siano $X_1, X_2, \ldots, X_n$ variabili casuali binarie indipendenti tali che $\mathbf{Pr}[X_i = 1] = p_i$ con $p_i < 1$ e sia $X = \sum_{i=1}^{n} X_i$ la variabile casuale somma e $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} p_i$ il suo valore atteso. Allora, per ogni $0 < \delta \leq 1$ abbiamo $\mathbf{Pr}[X < (1-\delta)\mu] < F^-(\mu, \delta)$ con*

$$F^-(\mu, \delta) = e^{-\frac{\mu\delta^2}{2}}$$

I due teoremi precedenti ci forniscono i limiti $F^+(\mu, \delta)$ ed $F^-(\mu, \delta)$ alla probabilità che X sia rispettivamente maggiore di $(1+\delta)\mu$ o minore di $(1-\delta)\mu$.

Talvolta noi siamo interessati al problema inverso: data una probabilità ϵ trovare un valore di δ tale che X risulti rispettivamente maggiore di $(1+\delta)\mu$ o minore di $(1-\delta)\mu$ con probabilità minore di ϵ.

Per rispondere a domande di questo tipo dobbiamo trovare due valori $\Delta^+(\mu,\epsilon)$ e $\Delta^-(\mu,\epsilon)$ della variabile δ tali che $F^+(\mu,\Delta^+(\mu,\epsilon)) \leq \epsilon$ ed $F^-(\mu,\Delta^-(\mu,\epsilon)) \leq \epsilon$.

Trovare $\Delta^-(\mu,\epsilon)$ non è difficile: basta esplicitare δ nell'equazione $F^-(\mu,\delta) = \epsilon$ ottenendo

$$\Delta^-(\mu,\epsilon) = \sqrt{\frac{2\ln(1/\epsilon)}{\mu}} \tag{1}$$

Meno facile è trovare $\Delta^+(\mu,\epsilon)$ in quanto l'equazione $F^+(\mu,\delta) = \epsilon$ non è facilmente esplicitabile in δ. Si conoscono comunque i seguenti limiti (di cui tralasciamo la dimostrazione) per $\Delta^+(\mu,\epsilon)$ e che valgono uno per valori piccoli di δ e l'altro per valori grandi:

- Se $\delta \leq 2e-1 = 4.4365\ldots$

$$\Delta^+(\mu,\epsilon) < \sqrt{\frac{4\ln(1/\epsilon)}{\mu}} \tag{2}$$

- Se $\delta > 2e-1$

$$\Delta^+(\mu,\epsilon) < \frac{\log_2(1/\epsilon)}{\mu} - 1 \tag{3}$$

Notiamo che i limiti di Chernoff che abbiamo visto si applicano alla somma di variabili casuali binarie *indipendenti*. Non possiamo quindi applicarli all'esempio dei marinai e delle brande.

Per illustrare tali limiti usiamo quindi un altro esempio.

Esempio 2.5 Due amici, Aldo e Bruno, giocano spesso a scacchi. Aldo è però meno bravo di Bruno per cui la probabilità che egli vinca una partita è soltanto $1/3$, la metà della probabilità $2/3$ che ha Bruno di vincere.

Si vuole calcolare un limite superiore alla probabilità che in un torneo di n partite Aldo ne vinca almeno la metà.

Sia $X_i = 1$ se Aldo vince la i-esima partita e $X_i = 0$ altrimenti. Il numero di partite vinte da Aldo è quindi $X = \sum_{i=1}^{n} X_i$. Poiché $\mathbf{E}[X_i] = 1/3$ abbiamo

$$\mu = \mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i] = n/3$$

Siccome le variabili X_i sono indipendenti possiamo usare il Teorema 2.3 con $\delta = 1/2$ per ottenere

$$\mathbf{Pr}[X > n/2] = \mathbf{Pr}[X > (1+\delta)\mu] < F^+(\mu, \delta) = 0.49^n$$

Dunque, ad esempio, la probabilità che su 6 partite Aldo ne vinca almeno 3 è minore di $0.49^6 = 0.014$.

Sia Y il numero di partite vinte da Bruno. Possiamo usare il Teorema 2.4 per calcolare un limite superiore alla probabilità che Bruno vica meno della metà delle partite. Il valore atteso del numero di partite vinte da Bruno è $\mu = 2n/3$ e usando il Teorema 2.4 con $\delta = 1/4$ otteniamo

$$\mathbf{Pr}[Y < n/2] = \mathbf{Pr}[Y < (1-\delta)\mu] < F^-(\mu, \delta) = 0.98^n$$

Possiamo osservare che $\mathbf{Pr}[X > n/2] = \mathbf{Pr}[Y < n/2]$ in quanto $X + Y = n$. Questo mostra che il secondo limite 0.98^n non è per niente stretto. In realtà neppure il primo limite 0.49^n è stretto ma entrambi decrescono esponenzialmente e quindi mostrano che quando n cresce entrambe tali probabilità tendono molto rapidamente a zero.

3 Rendering

In grafica computazionale occorre spesso proiettare una stessa scena da una successione di punti di vista differenti. In questo caso uno dei problemi è quello della eliminazione delle linee nascoste (che ovviamente cambiano a seconda del punto di vista). Può essere utile spendere un certo tempo per preelaborare la scena in modo da velocizzare le proiezioni.

Illustreremo uno dei modi per fare ciò: la costruzione di una partizione binaria della scena. Per semplicità di esposizione noi considereremo soltanto il caso di una scena bidimensionale costituita da alcuni segmenti complanari e che viene osservata da punti appartenenti allo stesso piano.

Problema 1 (Eliminazione linee nascoste) *Dato un insieme S costituito da n segmenti complanari $s_1, s_2, \ldots, s_n$ ed un punto di osservazione O appartenente allo stesso piano costruire la proiezione dal punto di osservazione O dell'insieme di segmenti eliminando le parti che risultano non visibili in quanto coperte da altri segmenti.*

Nella Figura 1 è illustrata la situazione con un insieme di quattro segmenti.

La stessa tecnica si potrà usare per risolvere la versione più interessante del problema in cui si richiede di proiettare su di un piano un oggetto tridimensionale costituito da componenti geometriche sia monodimensionali (segmenti) sia bidimensionali (poligoni, cerchi, ellissi, ecc.).

Un metodo per disegnare una scena che è ben noto ai pittori consiste nel disegnare dapprima gli oggetti più distanti dal punto di osservazione O e poi disegnare sopra gli oggetti più vicini. In questo modo le parti nascoste vengono automaticamente coperte.

Noi useremo lo stesso metodo poiché dopotutto uno schermo di computer si comporta come la tela del pittore: quando disegniamo un oggetto geometrico sul video noi modifichiamo i pixel corrispondenti cancellando la parte degli oggetti disegnati precedentemente che risulta coperta dal nuovo oggetto.

Dobbiamo quindi semplicemente ordinare i segmenti $s_1, s_2, \ldots, s_n$ in modo tale che se il segmento s_i copre in tutto o in parte il segmento s_j esso venga dopo di s_j in tale ordine.

Vi è però un problema che si può notare osservando i due segmenti s_3 ed s_4 nella Figura 1: ciascuno di essi copre una parte dell'altro e quindi non sappiamo quale disegnare per primo. Questo succede soltanto quando due segmenti si intersecano internamente (ossia quando il punto di intersezione non è un estremo dell'uno o dell'altro segmento).

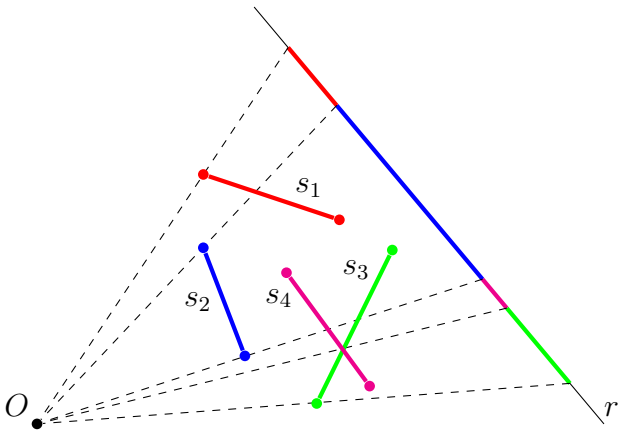

Figura 1: Una scena costituita da quattro segmenti complanari s_1, s_2, s_3 ed s_4 e le loro proiezioni sulla retta r dal punto di osservazione O. Per un osservatore in O parte dei segmenti s_1, s_3 ed s_4 è nascosta dal segmento s_2, parte del segmento s_3 è nascosta dal segmento s_4 e parte del segmento s_4 è nascosta dal segmento s_3.

La soluzione di questo problema consiste semplicemente nello spezzare uno dei due segmenti in due parti in corrispondenza del punto di intersezione. Per semplicità noi assumeremo che l'insieme $s_1, s_2, \ldots, s_n$ non contenga due segmenti che si intersecano internamente. Vedremo in seguito come questa restrizione possa essere rimossa.

Un problema meno grave si ha quando nessuno dei due segmenti copre in tutto o in parte l'altro come succede ad esempio per i segmenti s_1 ed s_3 di Figura 1: in questo caso l'ordine tra i due segmenti è irrilevante e noi sceglieremo arbitrariamente quale dei due deve venire prima.

Per decidere l'ordine, rispetto al punto di osservazione O, tra due segmenti s_1 ed s_2 che non si intersecano internamente procederemo nel seguente modo.

Consideriamo le due rette $\ell(s_1)$ ed $\ell(s_2)$ che contengono i segmenti s_1 e s_2. Deve allora accadere che o s_1 è tutto contenuto in uno dei due semipiani in cui la retta $\ell(s_2)$ divide il piano oppure s_2 è tutto contenuto in uno dei due semipiani in cui la retta $\ell(s_1)$ divide il piano (altrimenti le due rette dovrebbero intersecarsi in un punto che è interno sia ad s_1 che ad s_2 il che è impossibile dato che s_1 ed s_2 non si intersecano internamente).

Per fissare le idee supponiamo che s_2 sia tutto contenuto in uno dei due semipiani in cui la retta $\ell(s_1)$ divide il piano.

Per decidere quale dei due segmenti precede l'altro basta controllare in quale

semipiano si trova il punto di osservazione O: se si trova nello stesso semipiano che contiene s_2 il segmento s_1 deve precedere il segmento s_2, altrimenti è il segmento s_2 che deve precedere il segmento s_1.

Rimane da decidere come verificare se i due estremi di s_2 stanno nello stesso semipiano rispetto alla retta $\ell(s_1)$ e in caso affermativo se il punto di osservazione O sta nello stesso semipiano o nel semipiano opposto.

Per fare questo possiamo usare la procedura ANGLE-LEFT.

Il Programma 3 è lo pseudo codice della procedura PRECEDE che determina quale dei due segmenti $s_1 = \overline{p_1q_1}$ ed $s_2 = \overline{p_2q_2}$ debba essere disegnato per primo.

```
PRECEDE(s_1, s_2, O)    // s_1 = p_1q_1 ed s_2 = p_2q_2 ed O punto di osservazione.
 1  d_1 = ANGLE-LEFT(p_1, q_1, p_2)
 2  d_2 = ANGLE-LEFT(p_1, q_1, q_2)
 3  if (d_1 ≥ 0 and d_2 ≥ 0) or (d_1 ≤ 0 and d_2 ≤ 0)
 4       // p_2 e q_2 stanno nello stesso semipiano rispetto alla retta ℓ(s_1).
          // Il segmento s_1 precede s_2 se e solo se il punto di
          // osservazione O appartiene allo stesso semipiano di p_2 e q_2.
 5       d_3 = ANGLE-LEFT(p_1, q_1, O)
 6       return (d_1 ≥ 0 and d_2 ≥ 0 and d_3 ≥ 0) or
                    (d_1 ≤ 0 and d_2 ≤ 0 and d_3 ≤ 0)
 7  else // p_2 e q_2 non stanno nello stesso semipiano rispetto ad ℓ(s_1).
          // Dunque p_1 e q_1 devono stare nello stesso semipiano rispetto alla
          // retta ℓ(s_2). Il segmento s_1 precede s_2 se e solo se il punto
          // di osservazione O non appartiene allo stesso semipiano di p_1 e q_1.
 8       d_1 = ANGLE-LEFT(p_2, q_2, p_1)
 9       d_2 = ANGLE-LEFT(p_2, q_2, q_1)
10       d_3 = ANGLE-LEFT(p_2, q_2, O)
11       return (d_1 < 0 or d_2 < 0 or d_3 < 0) and
                    (d_1 > 0 or d_2 > 0 or d_3 > 0)
```

Programma 3: Il test di precedenza tra i segmenti s_1 ed s_2 rispetto all'origine O.

La procedura PRECEDE ci permette di controllare in tempo costante $O(1)$ quale tra due segmenti s_1 ed s_2 debba essere disegnato per primo. Possiamo quindi, in una fase di preelaborazione, ordinare i segmenti in tempo $O(n \log n)$ dopo di che la proiezione richiede tempo $O(n)$.

La soluzione che abbiamo trovato non è però molto soddisfacente nel caso in cui si debba calcolare la proiezione dello stesso insieme di segmenti da un punto

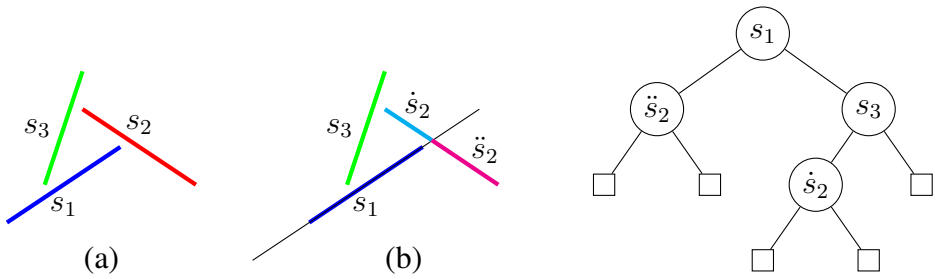

Figura 2: (a) Un insieme di segmenti per il quale non esiste un albero di partizione binaria: qualunque segmento si scelga come radice vi è sempre un segmento che non è tutto contenuto in uno dei due semipiani. In (b) il segmento s_2 è stato spezzato nelle due parti $\dot{s}_2$ ed $\ddot{s}_2$ in modo tale che l'insieme dei quattro segmenti $s_1, \dot{s}_2, \ddot{s}_2, s_3$ ammetta l'albero di partizione binaria disegnato alla destra.

di osservazione O mobile. In questo caso infatti dovremmo effettuare un nuovo riordino dei segmenti per ogni nuova posizione del punto di osservazione.

Ci si può chiedere se sia possibile effettuare una preelaborazione dell'insieme di segmenti che sia indipendente dal punto di osservazione e permetta ugualmente di effettuare la proiezione in tempo $O(n)$.

La risposta è affermativa e consiste nel costruire un particolare albero binario, detto *albero di partizione binaria*, in cui ogni nodo rappresenta un segmento s il quale suddivide il piano in due semipiani mediante la retta $\ell(s)$ che lo contiene.

L'albero di partizione binaria è ordinato in modo tale che il sottoalbero sinistro contenga soltanto segmenti che appartengono al semipiano negativo rispetto al segmento memorizzato nella radice mentre nel sottoalbero destro vi sono soltanto segmenti che appartengono al semipiano positivo.

Una volta che i segmenti siano stati inseriti in un albero di partizione binaria la proiezione di tali segmenti da un arbitrario punto di osservazione O si può effettuare in tempo $O(n)$ mediante la procedura ricorsiva che ha come parametri un puntatore x alla radice dell'albero di partizione binaria ed il punto di osservazione O. Lo pseudo codice è riportato nel Programma 4.

La procedura ricorsiva PROIETTA richiede tempo $O(n)$ per proiettare un insieme di n segmenti da un qualsiasi punto di osservazione O.

Purtroppo, come mostra la Figura 2(a), non tutti gli insiemi di segmenti possono essere rappresentati mediante un albero di partizione binaria. Possiamo però ovviare a questo inconveniente spezzando alcuni dei segmenti come si vede in Figura 2(b).

```
PROIETTA(x, O)   // x radice dell'albero, O punto di osservazione.
 1  if x ≠ NIL
 2      // x nodo radice con etichetta il segmento x.s = pq.
 3      d = TURN-LEFT(p, q, O)
 4      if d ≥ 0
 5          // O appartiene al semipiano positivo.
 6          PROIETTA(x.left, O)
 7          DISEGNA(x.s)
 8          PROIETTA(x.right, O)
 9      else // O appartiene al semipiano negativo.
10          PROIETTA(x.right, O)
11          DISEGNA(x.s)
12          PROIETTA(x.left, O)
```

Programma 4: La procedura per la proiezione dei segmenti contenuti nel sottoalbero di radice x rispetto al punto di osservazione O.

La procedura generale per ottenere un albero di partizione binario per un insieme qualsiasi $S = s_1, s_2, \ldots, s_n$ di segmenti è il Programma 5.

Notiamo che per la procedura PARTIZIONE-BINARIA non è necessario che i segmenti in input non si intersechino internamente. Se s_i ed s_j si intersecano internamente e $i < j$ la procedura PARTIZIONE-BINARIA quando prende in considerazione il segmento s_i usa la retta $\ell(s_i)$ per spezzare s_j.

Naturalmente, quando si spezzano dei segmenti, il loro numero aumenta e quindi aumenta il numero di nodi dell'albero che si ottiene. Siccome il tempo richiesto per eseguire la procedura PROIETTA è proporzionale al numero di nodi dell'albero è importante costruire un albero con meno nodi possibile.

Il numero di suddivisioni effettuate dalla procedura PARTIZIONE-BINARIA e quindi il numero di nodi dell'albero della partizione binaria dipende dell'ordine in cui vengono presi in considerazione i segmenti (vedi Figura 3). Occorre quindi scegliere opportunamente tale ordine.

Si conoscono algoritmi deterministici che costruiscono una partizione binaria di un insieme di n segmenti in modo tale che l'albero ottenuto abbia $O(n \log n)$ nodi ma questi algoritmi sono molto complicati e richiedono un tempo di esecuzione molto alto.

Recentemente è stato dimostrato il limite inferiore $\Omega(n \log n / \log \log n)$ per il numero di nodi dell'albero di partizione binaria nel caso pessimo (ossia per ogni algoritmo deterministico di partizione binaria esiste sempre un insieme di

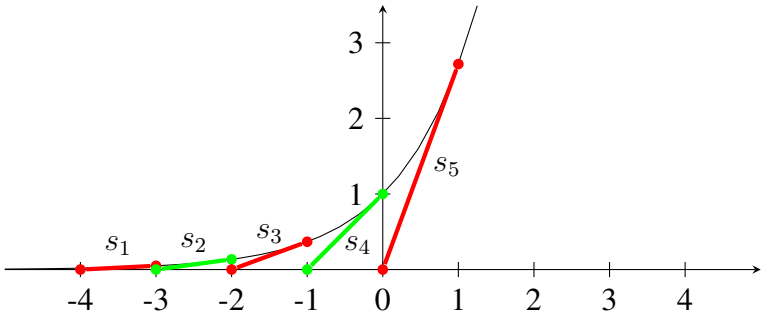

Figura 3: Un insieme di segmenti con estremi di coordinate $p = (x - 1, 0)$ e $q = (x, e^x)$. Se per effettuare la partizione binaria prendiamo in considerazione i segmenti per valori di x crescenti ogni segmento suddivide tutti i segmenti successivi per cui alla fine abbiamo un insieme di $n(n-1)/2 = \Theta(n^2)$ segmenti. Se li prendiamo in considerazione in ordine inverso non viene fatta nessuna suddivisione e dunque i segmenti rimangono n.

n segmenti tali che l'albero di partizione binaria calcolato dall'algoritmo abbia almeno $\Omega(n \log n / \log \log n)$ nodi). Non si sa se questo sia un limite stretto, ossia non si conosce ancora nessun algoritmo deterministico che raggiunga tale limite.

Invece di cercare un algoritmo deterministico che calcoli l'albero di partizione binaria minimo noi svilupperemo un semplice algoritmo randomizzato che dati n segmenti costruisce un albero di partizione binaria con valore atteso del numero di nodi $O(n \log n)$.

L'algoritmo randomizzato è il Programma 6.

Esso si limita a permutare casualmente i segmenti in ingresso dopo di che richiama l'algoritmo PARTIZIONE-BINARIA.

Dimostreremo ora il seguente teorema.

Teorema 3.1 *Il valore atteso del numero di nodi dell'albero di partizione binaria costruito dall'algoritmo randomizzato precedente è* $O(n \log n)$.

Dimostrazione. Dati due segmenti s ed s' tali che $\ell(s)$ tagli s' diciamo che s taglia s' a distanza $d(s, s') = k$ se partendo da s e muovendosi lungo la retta $\ell(s)$ verso s' si incontrano altri $k-1$ segmenti che vengono tagliati da $\ell(s)$. Per convenzione poniamo $d(s, s') = \infty$ quando $\ell(s)$ non taglia s'.

Siccome il segmento s si può estendere in entrambe le direzioni è possibile che $d(s, s') = d(s, s'')$ per due distinti segmenti s' ed s'' (in Figura 4 $d(s, s') = d(s, s'') = 2$).

```
PARTIZIONE-BINARIA(S, n)   // S = s_1, s_2, ..., s_n
 1  if n == 0
 2       return NIL
    // Altrimenti dividi il piano in due semipiani con la retta ℓ(s_1).
 3  npos = nneg = 0
 4  for i = 2 to n
 5       if "s_i è tutto nel semipiano positivo"
 6            npos = npos + 1, Spos[npos] = s_i
 7       elseif "s_i è tutto nel semipiano negativo"
 8            nneg = nneg + 1, Sneg[nneg] = s_i
 9       else "Dividi s_i in due parti ṡ_i e s̈_i usando la retta ℓ(s_1)"
10            npos = npos + 1, Spos[npos] = ṡ_i
11            nneg = nneg + 1, Sneg[nneg] = s̈_i
12  x.s = s_1
13  x.left = PARTIZIONE-BINARIA(Sneg, nneg)
14  x.right = PARTIZIONE-BINARIA(Spos, npos)
15  return x
```

Programma 5: La procedura per la costruzione della partizione binaria.

```
RANDOM-PARTIZIONE-BINARIA(S, n)
1  "Riordina casualmente i segmenti."
2  return PARTIZIONE-BINARIA(S, n)
```

Programma 6: La procedura randomizzata per la costruzione della partizione binaria.

Supponiamo che $\ell(s)$ tagli s' a distanza $d(s, s') = k$ e siano $s_1, s_2, \ldots, s_{k-1}$ i segmenti che la retta $\ell(s)$ taglia prima di tagliare s'. Durante l'esecuzione dell'algoritmo randomizzato il segmento s' può essere spezzato con la retta $\ell(s)$ soltanto se, nell'ordine in cui i segmenti vengono visitati dall'algoritmo, il segmento s viene prima dei k segmenti $s_1, s_2, \ldots, s_{k-1}, s'$. Siccome l'ordine è casuale la probabilità che ciò accada è $1/(k+1)$. Di conseguenza la probabilità che l'algoritmo randomizzato spezzi il segmento s' usando la retta $\ell(s)$ è minore o uguale di $1/(k+1)$ (questa formula vale anche se $\ell(s)$ non taglia s'; in questo caso infatti la formula fornisce la probabilità $1/\infty = 0$).

Sia $X_{s,s'}$ una variabile indicatrice che è 1 se durante l'esecuzione dell'algoritmo randomizzato il segmento s' viene spezzato con la retta $\ell(s)$ mentre vale 0

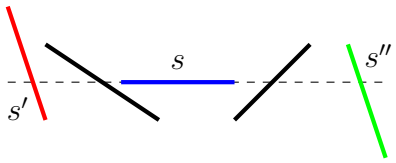

Figura 4: Un esempio in cui $d(s, s') = d(s, s'') = 2$.

altrimenti; chiaramente $\mathbf{E}[X_{s,s'}] \leq 1/(d(s, s') + 1)$.

Il numero m di nodi dell'albero di partizione binaria costruito dall'algoritmo è pari al numero n di segmenti in input più il numero di tagli effettuati, ossia

$$m = n + \sum_{s} \sum_{s' \neq s} X_{s,s'}$$

Quindi il valore atteso è

$$\mathbf{E}[m] = \mathbf{E}\left[n + \sum_{s} \sum_{s' \neq s} X_{s,s'}\right] = n + \sum_{s} \sum_{s' \neq s} \mathbf{E}[X_{s,s'}] \leq n + \sum_{s} \sum_{s' \neq s} \frac{1}{d(s, s') + 1}$$

Siccome per ogni segmento s e ogni intero positivo k vi sono al più 2 segmenti s' ed s'' per i quali $d(s, s') = d(s, s'') = k$ possiamo scrivere

$$\sum_{s' \neq s} \frac{1}{d(s, s') + 1} \leq \sum_{k=1}^{n-1} \frac{2}{k + 1} \leq 2H_n$$

dove $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$ è l'n-esimo numero armonico.

Si ottiene quindi

$$\mathbf{E}[m] \leq n + 2nH_n$$

e ricordando che $H_n = \ln n + O(1)$ si conclude che il valore atteso del numero di nodi dell'albero è $O(n \log n)$. □

Notiamo che il Teorema 3.1 ci assicura implicitamente che esiste sempre un albero di partizione binaria con al più $O(n \log n)$ nodi (ma non ci fornisce un modo per trovarlo).

Dettagli implementativi

Nella procedura PARTIZIONE-BINARIA viene richiesto di suddividere un segmento $s = \overline{p_1 p_2}$ in due parti $\dot{s}$ e $\ddot{s}$ usando la retta $\ell(s')$ che contiene il segmento $s' = \overline{p_3 p_4}$.

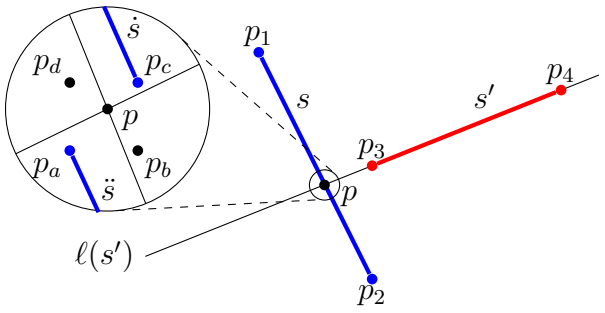

Figura 5: Suddivisione del segmento $s = \overline{p_1 p_2}$ in due parti $\dot{s}$ e $\ddot{s}$ usando la retta $\ell(s')$ che contiene il segmento $s' = \overline{p_3 p_4}$. Nell'ingrandimento i punti p_a, p_b, p_c e p_d sono i punti con coordinate intere più vicini al punto di intersezione p. Sono stati scelti i due punti $\dot{p} = p_c$ e $\ddot{p} = p_a$ per costruire i due segmenti $\dot{s} = \overline{p_1 \dot{p}}$ e $\ddot{s} = \overline{\ddot{p} p_2}$. Notiamo che i tre segmenti s, $\dot{s}$ e $\ddot{s}$ non sono collineari e nemmeno paralleli ma si avvicinano a tali condizioni nei limiti dell'approssimazione effettuata.

Per fare questo dobbiamo trovare il punto di intersezione p le cui coordinate possono non essere intere. La Figura 5 illustra la situazione.

La cosa migliore che possiamo fare è suddividere il segmento $s = \overline{p_1 p_2}$ nei due segmenti $\dot{s} = \overline{p_1 \dot{p}}$ e $\ddot{s} = \overline{\ddot{p} p_2}$ dove $\dot{p}$ e $\ddot{p}$ sono i punti di coordinate intere più vicini a p rispettivamente nel semipiano contenente p_1 e in quello contenente p_2.

In coordinate proiettive i coefficienti delle equazioni $ax + by + cz = 0$ e $a'x + b'y + c'z = 0$ delle rette $\ell(s)$ ed $\ell(s')$ sono

$$\begin{array}{rclcrcl} a & = & y_1 z_2 - y_2 z_1 & \qquad & a' & = & y_3 z_4 - y_4 z_3 \\ b & = & x_1 z_2 - x_2 z_1 & & b' & = & x_3 z_4 - x_4 z_3 \\ c & = & x_1 y_2 - x_2 y_1 & & c' & = & x_3 y_4 - x_4 y_3 \end{array}$$

e le coordinate proiettive (x, y, z) del punto p di intersezione sono

$$x = bc' - b'c, \quad y = ac' - a'c, \quad z = ab' - a'b$$

Se $z = 0$ i due segmenti sono paralleli. Siccome nel nostro caso sappiamo che p_1 e p_2 non stanno nello stesso semipiano rispetto alla retta $\ell(s')$ i due segmenti non possono essere paralleli e quindi $z \neq 0$.

Per ritornare a coordinate intere estese dobbiamo usare il coefficiente di proporzionalità per ridurre z a 1. Dobbiamo quindi dividere per z. Arrotondando le

divisioni una volta per eccesso e una volta per difetto troviamo le coordinate

$$\begin{array}{rclcrcl} \dot{x} & = & \lfloor (bc' - b'c)/(ab' - a'b) \rfloor & \qquad & \ddot{x} & = & \lceil (bc' - b'c)/(ab' - a'b) \rceil \\ \dot{y} & = & \lfloor (ac' - a'c)/(ab' - a'b) \rfloor & & \ddot{y} & = & \lceil (ac' - a'c)/(ab' - a'b) \rceil \end{array}$$

dei vertici del quadrato di lato unitario che contiene il punto p.

I quattro vertici sono quindi $p_a = (\dot{x}, \dot{y}, 1)$, $p_b = (\ddot{x}, \dot{y}, 1)$, $p_c = (\ddot{x}, \ddot{y}, 1)$ e $p_d = (\dot{x}, \ddot{y}, 1)$.

Sceglieremo quindi tra questi quattro vertici un punto $\dot{p}$ che stia nello stesso semipiano di p_1 ed un punto $\ddot{p}$ che stia nello stesso semipiano di p_2. In questo modo otteniamo i due segmenti $\dot{s} = \overline{p_1\dot{p}}$ e $\ddot{s} = \overline{\ddot{p}p_2}$.

Lo pseudo codice di questa procedura è riportato nel Programma 7.

```
DIVIDI-SEGMENTO(s, s')   // s = p1p2, s' = p3p4
 1  a = y1z2 − y2z1, b = x1z2 − x2z1, c = x1y2 − x2y1
 2  a' = y3z4 − y4z3, b' = x3z4 − x4z3, c' = x3y4 − x4y3
 3  ẋ = ⌊(bc' − b'c)/(ab' − a'b)⌋, ẏ = ⌊(ac' − a'c)/(ab' − a'b)⌋
 4  ẍ = ⌈(bc' − b'c)/(ab' − a'b)⌉, ÿ = ⌈(ac' − a'c)/(ab' − a'b)⌉
 5  pa = (ẋ, ẏ, 1), pb = (ẍ, ẏ, 1), pc = (ẍ, ÿ, 1), pd = (ẋ, ÿ, 1)
 6  d1 = ANGLE-LEFT(p3, p4, p1), d2 = ANGLE-LEFT(p3, p4, p2)
 7  da = ANGLE-LEFT(p3, p4, pa), db = ANGLE-LEFT(p3, p4, pb)
 8  dc = ANGLE-LEFT(p3, p4, pc), dd = ANGLE-LEFT(p3, p4, pd)
 9  if (d1 ≥ 0 and da ≥ 0) or (d1 ≤ 0 and da ≤ 0) then ṗ = pa
10  elseif (d1 ≥ 0 and db ≥ 0) or (d1 ≤ 0 and db ≤ 0) then ṗ = pb
11  elseif (d1 ≥ 0 and dc ≥ 0) or (d1 ≤ 0 and dc ≤ 0) then ṗ = pc
12  else ṗ = pd
13  if (d2 ≥ 0 and da ≥ 0) or (d2 ≤ 0 and da ≤ 0) then p̈ = pa
14  elseif (d2 ≥ 0 and db ≥ 0) or (d2 ≤ 0 and db ≤ 0) then p̈ = pb
15  elseif (d2 ≥ 0 and dc ≥ 0) or (d2 ≤ 0 and dc ≤ 0) then p̈ = pc
16  else p̈ = pd
17  ṡ = p1ṗ, s̈ = p̈p2
18  return ṡ, s̈
```

Programma 7: La procedura per la suddivisione del segmento s con la retta $\ell(s')$.

Esercizio 3 Cosa succede se uno o entrambi i parametri s ed s' della procedura DIVIDI-SEGMENTO sono illimitati, ossia sono delle semirette con uno dei due estremi all'infinito? Quali modifiche occorre apportare a DIVIDI-SEGMENTO perché gestisca correttamente anche questo caso?

4 Instradamento dei messaggi in un calcolatore parallelo

Il problema che affronteremo è quello della comunicazione in una rete di processori paralleli.

Useremo i limiti di Chernoff per dimostrare che l'algoritmo randomizzato da noi proposto risulta più efficiente di ogni possibile algoritmo deterministico.

Rappresentiamo una rete di n processori paralleli mediante un grafo orientato i cui vertici rappresentano i processori mentre gli archi rappresentano i canali di trasmissione tra i processori.

La comunicazione tra i processori avviene in modo sincrono. Ad intervalli di tempo regolari viene eseguito un *passo di trasmissione* in cui ogni processore può inviare un solo messaggio unitario (pacchetto) su ciascuno dei suoi canali di output e può ricevere un solo messaggio unitario da ciascuno dei suoi canali di input.

Talvolta un processore potrebbe ricevere contemporaneamente dai suoi canali di input più pacchetti da inviare sullo stesso canale di output. In questo caso i pacchetti vengono immagazzinati in una coda associata al canale di output in modo da poter essere spediti uno alla volta mentre gli altri rimangono fermi nella coda per un certo numero di passi di trasmissione accumulando del ritardo.

Supponiamo che i processori siano numerati da 1 ad n. Ogni pacchetto p che venga trasmesso attraverso la rete parte da un processore i di origine ed è diretto ad un processore d di destinazione. Compito di un algoritmo di instradamento è scegliere il percorso più conveniente per far arrivare il pacchetto p al processore d nel più breve tempo possibile.

La scelta che sembra più ovvia è scegliere il cammino minimo da i a d (ci sono algoritmi efficienti in grado di calcolare i cammini minimi).

Purtroppo quando nella rete girano molti pacchetti contemporaneamente occorre tener conto della congestione della rete, ossia dei ritardi dovuti ai tempi in cui un messaggio rimane bloccato nella coda di un canale di output in attesa che arrivi il suo turno per essere trasmesso.

Consideriamo la seguente situazione: ad un certo istante iniziale ogni processore i invia un pacchetto p_i ad un processore di destinazione d_i. Per semplificare l'esposizione supponiamo che i processori di destinazione d_i siano tutti distinti. In questa situazione $d_1, \ldots, d_n$ è una permutazione di $1, \ldots, n$ e per questa ragione parleremo di *problema dell'instradamento di una permutazione*.

Il problema particolare dell'instradamento di una permutazione è importante per l'implementazione di modelli astratti di calcolo parallelo quali la PRAM (la RAM parallela).

Diremo che un algoritmo di instradamento è *ignaro* se calcola il percorso di un pacchetto soltanto in base alla sua origine ed alla sua destinazione e senza tener conto dei percorsi degli altri pacchetti.

Se si usa un algoritmo deterministico ignaro il ritardo dovuto alla congestione in corrispondenza delle code di output può risultare notevole. Vale infatti il seguente teorema (di cui omettiamo la dimostrazione):

Teorema 4.1 (Valiant) *Sia R una rete di n processori paralleli tale che ogni processore abbia al più k canali di output. Per ogni algoritmo deterministico ignaro esiste una scelta delle destinazioni $d_1, \ldots, d_n$ tale che il numero di passi di trasmissione per far arrivare a destinazione tutti gli n pacchetti sia $\Omega(\sqrt{n/k})$.*

Ad esempio, consideriamo il caso in cui la rete ha la struttura di un ipercubo. L'ipercubo a k dimensioni ha $n = 2^k$ vertici numerati da 0 a $n - 1$ e ogni vertice è etichettato con la rappresentazione binaria $i_0 \ldots i_{k-1}$ del suo numero d'ordine $i = i_0 2^{k-1} + i_1 2^{k-2} + \ldots + i_{k-2} 2 + i_{k-1}$. Due vertici i e j sono connessi con un lato se e solo se le due rappresentazioni binarie $i_0 \ldots i_{k-1}$ e $j_0 \ldots j_{k-1}$ differiscono in una sola posizione.

Ogni vertice dell'ipercubo rappresenta un processore e ad ogni lato dell'ipercubo corrispondono due canali di trasmissione, uno per ognuna delle due direzioni. Dunque ogni processore ha k canali di output e k canali di input e i canali sono in totale nk. Chiaramente in un ipercubo vi è sempre un cammino di lunghezza minore o uguale a k tra due processori qualsiasi. D'altra parte il teorema precedente ci dice che, nel caso peggiore, occorrono almeno $\sqrt{n/k}$ passi per trasmettere tutti gli n messaggi. Ad esempio per $k = 32$ abbiamo $\sqrt{2^{32}/32} = 11585 \gg 32$.

Un strategia di instradamento ignara molto naturale su di un ipercubo è quella *bit a bit* che consiste nel fissare un bit della destinazione alla volta. L'algoritmo bit a bit confronta i bit della destinazione con i bit del processore in cui si trova attualmente il pacchetto e invia il pacchetto sul canale corrispondente al primo bit diverso. In questo modo dopo aver processato i primi i bit il pacchetto si troverà nel processore la cui rappresentazione binaria ha i primi i bit uguali a quelli della destinazione e i successivi bit ancora uguali a quelli del processore di partenza.

Ad esempio se in un ipercubo di dimensione 4 il pacchetto si trova nel processore $11 = 1011_2$ e la sua destinazione è il processore $12 = 1100_2$ esso passerà per i processori intermedi $15 = 1111_2$ e $13 = 1101_2$.

Vediamo ora un semplice algoritmo ignaro randomizzato che con altissima probabilità richiede un numero di passi notevolmente inferiore ad $\Omega(\sqrt{n/k})$.

Algoritmo 2 *Per ciascuno degli n processori scegli una destinazione intermedia σ_i in modo casuale e indipendente. Usa l'algoritmo deterministico bit a bit per instradare ogni pacchetto p_i dal processore i al processore σ_i e usa quindi lo stesso algoritmo deterministico bit a bit per far proseguire il pacchetto p_i dal processore σ_i al processore di destinazione d_i.*

Nelle code di attesa viene data precedenza ai pacchetti che devono ancora arrivare alla destinazione intermedia.

L'algoritmo randomizzato si limita quindi ad usare l'algoritmo deterministico bit a bit per inviare dapprima i messaggi ad una destinazione intermedia scelta casualmente e quindi dalla destinazione intermedia alla destinazione finale.

Quanti passi sono necessari perché un pacchetto p_i spedito dal processore i arrivi alla sua destinazione d_i?

Consideriamo la prima fase in cui il pacchetto p_i segue un percorso ρ_i dal processore i al processore intermedio σ_i. Il percorso ρ_i è costituito da una sequenza $e_1, e_2, \ldots, e_m$ di canali, uno per ogni posizione in cui differiscono le rappresentazioni binarie di i e di σ_i. Siccome le rappresentazioni binarie di i e di σ_i hanno lunghezza k sarà sicuramente $m \leq k$.

Il numero di passi necessari per far arrivare p_i alla destinazione intermedia sarà quindi pari a m più il numero di passi in cui p_i deve rimane nella coda di attesa di un canale di output di qualche processore intermedio. Dobbiamo quindi valutare i ritardi dovuti ai tempi di attesa nelle code.

Il seguente lemma ci dice quali siano i processori comuni ai percorsi di due pacchetti p_i e p_j.

Lemma 4.2 *Siano p_i e p_j due pacchetti che partono dai processori i e j ed arrivano alle destinazioni intermedie σ_i e σ_j.*

Sia s l'ultima posizione in cui le rappresentazioni binarie di i e j differiscono e t la prima posizione in cui differiscono σ_i e σ_j (ponendo $t = k$ se $\sigma_i = \sigma_j$).

Se $s \geq t$ i due percorsi non hanno processori in comune. Se $s < t$ i due pacchetti stanno su processori distinti finchè l'algoritmo bit a bit processa i bit nelle posizioni da 0 ad s, stanno sugli stessi processori quando vengono processati i bit nelle posizioni da $s+1$ a $t-1$ e stanno su processori distinti quando vengono processati i bit nelle posizioni da t a $k-1$.

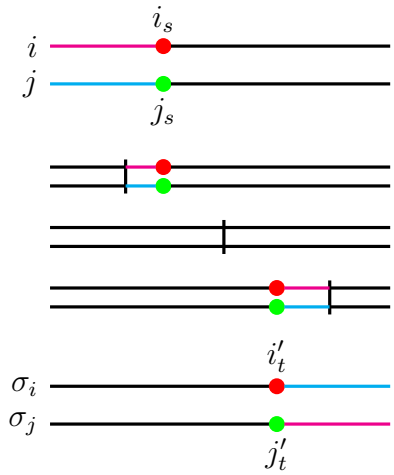

Figura 6: Illustrazione del Lemma 4.2. In alto le rappresentazioni binarie dei due processori di partenza i e j e in basso quelle dei processori di destinazione σ_i e σ_j. I bit delle parti colorate in nero sono uguali, i pallini rossi e verdi indicano due bit diversi e i bit delle parti colorate con colori differenti possono essere sia uguali che diversi. In mezzo sono le rappresentazioni dei processori intermedi dopo che l'algoritmo bit a bit ha rispettivamente processato i primi $k < s$, $s \leq k < t$ e $k \geq t$ bit sia di i che di j. Si osserva che per $k < s$ i due processori differiscono almeno per il bit in posizione s, per $s \leq k < t$ sono uguali e per $k \geq t$ differiscono almeno per il bit in posizione t.

Dimostrazione. Siano $i_0 \dots i_{k-1}$ e $j_0 \dots j_{k-1}$ le rappresentazioni binarie dei due processori di partenza i e j ed $i'_0 \dots i'_{k-1}$ e $j'_0 \dots j'_{k-1}$ le rappresentazioni binarie dei due processori di arrivo σ_i e σ_j. Sappiamo che $i_s \neq j_s$ e $i_{s+1} \dots i_{k-1} = j_{s+1} \dots j_{k-1}$ e che $i'_0 \dots i'_{t-1} = j'_0 \dots j'_{t-1}$ e $i'_t \neq j'_t$ (vedi Figura 6).

Finchè l'algoritmo bit a bit non ha processato il bit in posizione s i due pacchetti si trovano in processori diversi (la loro rappresentazione binaria differisce almeno per il bit in posizione s che non è stato ancora processato).

Se $s < t$, dopo aver processato il bit in posizione s e finchè non sia stato processato anche il bit in posizione t i due pacchetti si trovano negli stessi processori (i bit processati appartengono alla parte comune delle destinazioni σ_i e σ_j mentre gli ultimi appartengono ancora alla parte comune di i e j).

Infine, dopo aver processato il bit in posizione t i due pacchetti stanno su processori distinti (le rappresentazioni binarie dei due processori differiscono almeno per i bit in posizione t che sono uguali a quelli delle due destinazioni). □

Per il Lemma 4.2 i percorsi dei due pacchetti p_i e p_j possono avere in comune soltanto canali che congiungono due dei processori comuni in cui essi si trovano dopo che l'algoritmo bit a bit abbia processato il bit in posizione s e prima che esso abbia processato il bit in posizione t. Dunque i canali che il percorso $\rho_i = e_1, e_2, \ldots, e_m$ di un pacchetto p_i ha in comune con il percorso di un altro pacchetto sono consecutivi.

Supponiamo che al pacchetto p_i per percorrere il cammino $\rho_i = e_1, e_2, \ldots, e_m$ richieda $m+r$ passi accumulando un ritardo r. Il prossimo Lemma 4.3 attribuisce la causa di ciascuno degli r ritardi di p_i ad un pacchetto il cui cammino ha almeno un canale in comune con ρ_i.

L'idea è che se ad un certo istante t il pacchetto p_i rimane bloccato per almeno un passo di trasmissione nella coda di un canale e_h esso non potrà certamente trovarsi all'istante $t + k$ nella coda di un canale successivo e_{h+k}.

Inoltre se all'istante t il pacchetto p_i rimane bloccato ci deve essere un altro pacchetto $p(t, h)$ che all'istante t attraversa il canale e_h. All'istante successivo $t+1$ può accadere che il pacchetto $p(t, h)$ sia arrivato a destinazione oppure debba proseguire su di un canale che non appartiene al cammino ρ_i. In questo caso il cammino di $p(t, h)$ non ha altri canali in comune con ρ_i (per il Lemma 4.2) e quindi $p(t, h)$ non potrà essere causa di altri ritardi per p_i. Dunque attribuiremo al pacchetto $p = p(t, h)$ la causa del ritardo di p_i all'istante t.

Se $p(t, h)$ deve proseguire sul canale e_{h+1} allora all'istante $t + 1$ esso si trova nella coda del canale e_{h+1}. Siccome all'istante $t + 1$ la coda del canale e_{h+1} non è vuota vi è certamente un pacchetto $p(t + 1, h + 1)$ che attraversa il canale e_h all'istante $t+1$ (o il pacchetto $p(t, h)$ stesso o un'altro). Ancora, se $p(t+1, h+1)$ è arrivato a destinazione oppure deve proseguire su un canale che non appartiene al cammino ρ_i esso non può causare altri ritardi a p_i e quindi possiamo attribuire univocamente a $p = p(t + 1, h + 1)$ la causa del ritardo di p_i all'istante t.

Se anche $p(t + 1, h + 1)$ prosegue lungo il cammino ρ_i possiamo ripetere il ragionamento fino a trovare un pacchetto $p(t+k, h+k)$ che dopo aver percorso il canale e_{h+k} di ρ_i all'istante $t + k$ è arrivato a destinazione oppure deve proseguire su un canale che non appartiene al cammino ρ_i (dobbiamo trovarlo perchè $h+k \leq m$). Possiamo quindi attribuire univocamente a $p = p(t + k, h + k)$ la causa del ritardo di p_i all'istante t.

Abbiamo quindi provato il seguente lemma.

Lemma 4.3 *Sia $\rho_i = e_1, e_2, \ldots, e_m$ il percorso di un pacchetto p_i e supponiamo che al tempo t esso abbia attraversato i primi h canali del suo cammino e si trovi nella coda di attesa del canale e_{h+1} con un ritardo accumulato $\ell = t - h$.*

Se il pacchetto p_i nel passo successivo rimane bloccato nella coda allora, per qualche $k \geq 1$ deve esistere un altro pacchetto p che dopo aver percorso l'arco e_{h+k} del cammino ρ_i al tempo $t + k = h + k + \ell$, al tempo $t + k + 1$ esce dal cammino ρ_i (perché è arrivato a destinazione oppure perché deve proseguire su un canale che non appartiene al cammino ρ_i).

Per Lemma 4.3 per ogni $\ell = 0, \ldots, r-1$ se il pacchetto p_i dopo aver percorso il canale e_h rimane bloccato al tempo $t = h + \ell$ nella coda del canale e_{h+1} accumulando un ulteriore ritardo deve esistere un pacchetto che esce dal cammino dopo aver percorso un canale e_{h+k} al tempo $t + k = h + k + \ell$.

Siccome, per il Lemma 4.2, ogni pacchetto il cui cammino contiene un canale di ρ_i esce da ρ_i una sola volta i pacchetti relativi a ritardi ℓ diversi devono essere diversi.

Vale quindi il seguente corollario del Lemma 4.3.

Corollario 4.4 *Se il pacchetto p_i percorre il cammino $\rho_i = e_1, e_2, \ldots, e_m$ in tempo t accumulando un ritardo totale $r = t - m$ devono esserci almeno r altri pacchetti i cui cammini hanno almeno un canale in comune con ρ_i.*

Osserviamo che il corollario precedente fornisce una maggiorazione del ritardo r_i accumulato dal pacchetto p_i che dipende soltanto dai cammini percorsi dai pacchetti ma non dai tempi in cui tali pacchetti percorrono i rispettivi cammini e neppure dal tempo in cui ciascuno di essi parte dal processore iniziale. Quindi tale maggiorazione vale anche se i pacchetti partono in tempi diversi.

Per ogni coppia di pacchetti p_i e p_j introduciamo la variabile casuale indicatrice $X_{i,j}$ il cui valore è 1 se i due cammini ρ_i e ρ_j hanno almeno un canale in comune e 0 altrimenti.

Per il Lemma 4.4 il ritardo r_i accumulato dal pacchetto p_i è minore o uguale alla somma delle $X_{i,j}$ con $j \neq i$, ossia

$$r_i \leq \sum_{j \neq i} X_{i,j}$$

Per ogni canale e indichiamo con $P(e)$ il numero di pacchetti che transitano per e. Il ritardo r accumulato dal pacchetto p_i per percorrere il cammino $\rho_i = e_1, e_2, \ldots, e_m$ soddisfa la disuguaglianza

$$r_i \leq \sum_{j \neq i} X_{i,j} < \sum_{j=1}^{n} X_{i,j} \leq \sum_{i=1}^{m} P(e_i)$$

e considerando i valori attesi

$$\mathbf{E}[r_i] \leq \mathbf{E}\left[\sum_{j\neq i} X_{i,j}\right] < \mathbf{E}\left[\sum_{i=1}^{m} P(e_i)\right] = \sum_{i=1}^{m} \mathbf{E}[P(e_i)]$$

Calcoliamo quindi $\mathbf{E}[P(e)]$. La prima osservazione è che, per ragioni di simmetria, $\mathbf{E}[P(e)]$ è lo stesso per tutti i canali dell'ipercubo. Il valore atteso della lunghezza m di un cammino ρ_i è pari al numero di bit diversi tra le rappresentazioni binarie di i e σ_i. Siccome σ_i viene scelto casualmente ogni bit di σ_i ha probabilità $1/2$ di essere diverso dal corrispondente bit di i. Dunque il valore atteso di m è $k/2$ e il valore atteso della somma delle lunghezze dei cammini di tutti i pacchetti è $nk/2$. Il numero di canali dell'ipercubo è nk e quindi $\mathbf{E}[P(e)] = 1/2$ per ogni arco e. Dunque

$$\mathbf{E}[r_i] \leq \mathbf{E}\left[\sum_{j\neq i} X_{i,j}\right] < \sum_{i=1}^{m} \mathbf{E}[P(e_i)] = m/2 \leq k/2$$

Dunque ogni pacchetto arriva alla destinazione intermedia con un ritardo medio minore di $k/2$.

A noi interessa conoscere un limite per il tempo richiesto per far arrivare *tutti* i pacchetti alla destinazione intermedia. Useremo il limite di Chernoff del Teorema 2.3 applicato alla variabile casuale $X = \sum_{j\neq i} X_{i,j}$. Le variabili casuali $X_{i,j}$ sono indipendenti e $\mathbf{Pr}[X_{i,j} = 1] < 1$. Inoltre $\mu = \mathbf{E}[X] \leq k/2$. Possiamo quindi applicare il Teorema 2.3 ottenendo

$$\mathbf{Pr}[X > (1+\delta)\frac{k}{2}] \leq \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\frac{k}{2}}$$

per ogni $\delta > 0$. Se prendiamo, ad esempio, $\delta = 7$ otteniamo

$$\mathbf{Pr}[X > 4k] \leq \left[\frac{e^7}{8^8}\right]^{\frac{k}{2}} = 2^{\frac{k}{2}\log_2\left(\frac{e^7}{8^8}\right)} < 2^{-6k}$$

Siccome il numero totale di pacchetti è $n = 2^k$ la probabilità che almeno uno di essi abbia un ritardo maggiore di $4k$ è inferiore a $n2^{-6k} = 2^{-5k} = \frac{1}{32^k}$. Quindi tutti i pacchetti arrivano a destinazione in un tempo minore di $5k$ con probabilità maggiore di $1 - \frac{1}{32^k}$.

Cosa accade ai pacchetti nella seconda fase quando vanno dalla destinazione intermedia σ_i alla destinazione finale d_i. La situazione è analoga a quella della

prima fase con l'unica differenza che questa volta per ogni pacchetto p_i è determinato il processore di destinazione mentre viene scelto casualmente il processore di partenza. Un ragionamento analogo mostra che anche per la seconda fase il tempo richiesto per far arrivare tutti i pacchetti a destinazione risulta minore di $5k$ con probabilità maggiore di $1 - \frac{1}{32^k}$.

Supponiamo ora che alla fine della prima fase i pacchetti rimangano fermi in σ_i finché non siano arrivati a destinazione tutti i pacchetti e soltanto a quel punto inizi la fase due. In questo caso il tempo totale per far arrivare tutti i pacchetti dal punto di partenza i alla destinazione d_i sarà minore di $10k$ con probabilità maggiore di $1 - \frac{1}{32^k}$.

Possiamo quindi dimostrare il seguente

Teorema 4.5 *Usando l'algoritmo randomizzato tutti i pacchetti raggiungono la loro destinazione in meno di* $10k$ *passi con probabilità maggiore di* $1 - \frac{1}{32^k}$.

Dimostrazione. Per quello che abbiamo visto la cosa è certamente vera se i pacchetti iniziano la seconda fase contemporaneamente dopo che tutti i pacchetti abbiano completato la prima fase. Il problema è quindi considerare le possibili interferenze di pacchetti che stanno già eseguendo la seconda fase con pacchetti che stanno ancora eseguendo la prima fase. Ma la politica adottata nelle code per cui i pacchetti nella prima fase hanno precedenza sui pacchetti della seconda fase evita tali interferenze. □

Con l'ipercubo in $k = 32$ dimensioni l'algoritmo randomizzato manda tutti i pacchetti a destinazione in meno di 320 passi con probabilità maggiore di $1 - \frac{1}{32^{32}}$ (e quindi con certezza praticamente assoluta). Ricordando che l'algoritmo deterministico richiede nel caso peggiore 11585 passi è evidente il vantaggio di usare l'algoritmo randomizzato.

Esercizio 4 Dimostrare che se si usa l'algoritmo bit a bit su di un ipercubo a k dimensioni esiste una permutazione che richiede $\Omega(\sqrt{n})$ passi per far arrivare tutti i pacchetti a destinazione (più precisamente richiede almeno $\frac{1}{2}\sqrt{n} = 2^{\frac{k-2}{2}}$ passi se k è pari e almeno $\frac{1}{2\sqrt{2}}\sqrt{n} = 2^{\frac{k-3}{2}}$ passi se k è dispari).

Suggerimento: Sia $b_1 \dots b_k$ la rappresentazione binaria del processore di partenza i. Dividiamo la sequenza di bit in una parte iniziale x una parte centrale y e una parte finale z dove per k dispari la parte centrale è costituita dal solo carattere centrale mentre x e z hanno la stessa lunghezza $h = (k-1)/2$ mentre per k pari la parte centrale contiene i due caratteri centrali e le due parti laterali hanno lunghezza $h = k/2 - 1$.

Per ogni i scegliamo la destinazione d_i la cui rappresentazione binaria si ottiene da quella di i complementando i bit della parte centrale e scambiando le parti laterali. In altre parole per $i = x \cdot y \cdot z$ prendiamo $d_i = z \cdot \overline{y} \cdot x$