4
Algoritmi Randomizzati

L'idea di usare la casualità negli algoritmi può sembrare strana di primo acchito: già abbiamo spesso la percezione che i computer si comportino in maniera poco comprensibile (naturalmente ciò è dovuto a errori di vario tipo), vogliamo aggiungere ulteriori problemi con scelte casuali? In realtà le cose stanno diversamente e le scelte casuali possono essere di aiuto nella risoluzione di un problema. In questo capitolo vedremo come delle scelte casuali possano aiutarci a risolvere un problema che non è risolvibile in altro modo o a risolvere più efficientemente un problema che con un algoritmo deterministico richiederebbe una soluzione più costosa. Come spesso accade, ad un vantaggio corrisponde uno svantaggio. I benefici portati dalle scelte casuali, dovranno essere "pagati" con una piccola probabilità di insuccesso, che però può essere controllata e quindi assume valori accettabili in pratica. Inoltre l'insuccesso può anche essere non molto dannoso, come ad esempio impiegare il tempo che comunque sarebbe stato necessario nel caso pessimo.

4.1 Introduzione

Un algoritmo è definito come un processo deterministico (una sequenza di passi ben definiti) che a partire da un input produce un output, come mostrato nella Figura 4.1.

Figura 4.1: Algoritmo deterministico

Un'algoritmo deterministico produce sempre lo stesso output $y = f(x)$ su un dato input x. Per un algoritmo deterministico occorre provare che esso produce sempre la soluzione (corretta) e tipicamente si richiede che lo faccia in un tempo ragionevole (polinomiale nella grandezza dell'input).

Un algoritmo randomizzato è un algoritmo che fa delle scelte casuali: ha a disposizione un generatore di numeri casuali che può utilizzare durante l'esecuzione. La

Figura 4.2 rappresenta in modo schematico un algoritmo randomizzato.

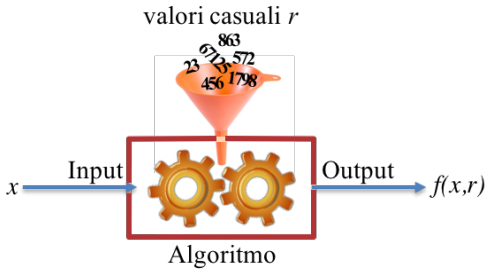

Figura 4.2: Algoritmo randomizzato

L'uso di valori casuali fa sì che l'output $y = f(x, r)$ non dipenda più esclusivamente dall'input x, come succede per il caso deterministico, ma anche dai valori casuali r utilizzati durante l'esecuzione.

Le scelte casuali possono migliorare il comportamento dell'algoritmo. Se si pensa all'analisi del caso medio e del caso pessimo si capisce il perchè. Un algoritmo deterministico può avere un comportamento molto inefficiente su dei particolari input. Tuttavia questi input che causano il comportamento molto inefficiente possono essere individuati proprio perchè l'algoritmo fa delle scelte deterministiche. Magari lo stesso algoritmo si comporta bene mediamente, e quindi solo in qualche raro caso manifesta un comportamento inefficiente. Fare delle scelte casuali ci permette di "mescolare" i dati in input. Questo fa sì che non si possa individuare a priori il caso pessimo e pertanto il comportamento atteso dell'algoritmo è quello del caso medio. Ci sono anche altre situazioni in cui la casualità è di aiuto. Ad esempio nei sistemi distribuiti, di cui parleremo successivamente, è utile per uscire da situazioni di impasse dovute alla simmetria del sistema.

4.1.1 Un esempio semplice

Consideriamo un esempio semplice: il gioco delle tre carte. In questo gioco chi distribuisce le carte, che per i nostri scopi assumeremo essere onesto, ha tre carte che vengono mischiate e posizionate di dorso sul tavolo. Delle tre carte due non hanno valore (es. due figure) mentre una sì (es. un asso). Lo scopo del gioco è quello di indovinare quale è la carta che ha valore. Per giocare potremo scegliere una, due o anche tre carte, ma ogni volta che decidiamo di scegliere una carta paghiamo 1€. Quando scopriamo la carta di valore otteniamo una vincita di 2€. Quindi se indoviniamo la carta con un solo tentativo abbiamo un guadagno di 1€, se la indoviniamo con due tentativi andiamo in pari, se la indoviniamo con tre tentativi perdiamo 1€. Supponiamo di dover giocare a questo gioco con un algoritmo che decide in quale ordine scoprire le carte. Si noti come in questo esempio l'input x è la posizione delle carte che può variare ad ogni partita in quanto viene decisa da chi mette le carte sul tavolo. L'algoritmo (deterministico) invece deve essere stabilito una volta per tutte e non potrà cambiare. Quindi, un algoritmo deterministico non può far altro che scegliere una determinata sequenza e chiedere di scoprire le carte in quell'ordine. Ad esempio un

algoritmo A_1 potrebbe chiedere di scoprire le carte in questo ordine: $2, 3, 1$. Un altro algoritmo A_2 potrebbe usare l'ordine $1, 3, 2$ (in questo esempio semplice esistono solo 6 diversi algoritmi deterministici). Qualunque sia l'algoritmo, che assumiamo sia noto a tutti quindi anche a chi distribuisce le carte, è possible fare in modo che l'algoritmo perda sempre 1€: basta posizionare la carta vincente nella posizione che l'algoritmo scoprirà per ultima. Poichè lo scopo di chi distribuisce le carte è quello di guadagnare soldi, piazzerà la carta vincente sempre nell'ultima posizione scelta dall'algoritmo deterministico. Pertanto il "guadagno" dell'algoritmo è una perdita netta di 1€ a partita. Quindi se giochiamo n partite perderemo n€.

Diamo adesso all'algoritmo la possibilità di fare delle scelte casuali. Grazie a questa capacità l'algoritmo ora non deve più stabilire a priori una determinata sequenza per scoprire le carte ma può decidere l'ordine con cui scopire le carte durante l'esecuzione, cioè *dopo* che le tre carte sono state sistemate sul tavolo. Adesso chi distribuisce le carte, che come prima conosce l'algoritmo ma non le scelte casuali perchè queste verranno fatte al momento dell'esecuzione, non può più sistemare la carta nell'ultima posizione che l'algoritmo chiederà di scoprire! Quale è adesso il guadagno atteso del giocatore? Si noti che mentre nel caso precedente, poichè veniva forzato ad ogni esecuzione il caso pessimo, in ogni partita l'algoritmo scopriva tutte e tre le carte. Ora c'è la possibilità che l'algoritmo scopra la carta vincente anche al primo o al secondo tentativo. Assumendo che la posizione della carta vincente sia scelta uniformemente fra le tre possibili, si ha che in media l'algoritmo indovinerà al primo tentativo $1/3$ delle volte e al secondo tentativo un altro $1/3$ delle volte (e ovviamente per il restante $1/3$ al terzo tentativo). Quindi il costo di m partite è di $\frac{m}{3} + \frac{2m}{3} + \frac{3m}{3} = 2m$. Poichè le vincite ammontano a $2m$, il guadagno atteso è 0.

Dunque, con l'algoritmo deterministico si ha una perdita netta di 1€ a partita, mentre con un algoritmo randomizzato si va in pari[1].

[1] Questo è un buon motivo per non giocare a questo gioco: se chi distribuisce le carte è onesto dobbiamo aspettarci di non guadagnare nulla.

4.1.2 *Un esempio più complesso*

Come altro esempio consideriamo il problema dell'ordinamento e analizziamo il comportamento dell'algoritmo QuickSort. Tale algoritmo divide il problema in due sottoproblemi in base a un elemento perno. Se la scelta di questo perno divide il problema, ad ogni chiamata ricosiva, in due sottoproblemi di pari grandezza, l'algoritmo risulta più efficiente e risolve il problema in $O(n \log n)$. Se invece la scelta del perno genera una suddivisione in due problemi molto sbilanciati allora l'algoritmo necessita di tempo $\Theta(n^2)$. Quindi la scelta del perno gioca un ruolo fondamentale. Usando delle scelte deterministiche, qualunque sia la scelta (es. il primo elemento, oppure l'ultimo, oppure quello centrale) sarà possibile fornire una sequenza di input che causa sempre un perno inefficiente. Se invece la scelta del perno viene fatta in modo casuale, allora non si potrà a priori determinare un input per cui l'algoritmo è inefficiente. Ovviamente l'algoritmo potrà comunque essere inefficiente, e magari adesso potrà esserlo su degli input per i quali la versione deterministica si comporta meglio, ma il punto cruciale è che adesso a causare il comportamento inefficiente sono esclusivamente le scelte casuali. E siccome l'algoritmo si comporta bene mediamente, il comportamento atteso della

versione randomizzata è quello efficiente. L'altro punto importante da notare e che è possibile fare una scelta casuale del perno in quanto l'algoritmo QUICKSORT funziona per qualunque scelta del perno. Quindi possiamo sceglierlo casualmente senza nessun problema. Fra poco vedremo più in dettaglio la versione randomizzata di QUICKSORT.

4.1.3 Vantaggi e svantaggi

Dunque, l'uso di scelte casuali toglie la possibilità a chi fornisce l'input di generare sempre il caso pessimo. Questo non è l'unico vantaggio che deriva dall'uso della casualità negli algoritmi. Ad esempio, nei sistemi distribuiti, l'uso della casualità da parte dei singoli processi può ridurre la quantità di informazioni da comunicare. La randomizzazione in questi casi è un utilissimo strumento per poter superare i problemi dovuti alla *simmetria* delle parti in gioco.

Come spesso accade c'è un rovescio della medaglia: il vantaggio ottenuto con la randomizzazione lo si paga con uno svantaggio che ovviamente deriva anche esso dall'uso della randomizzazione. Una classificazione tipica che si fa degli algoritmi randomizzati è in base alla natura dello svantaggio e gli algoritmi vengono partizionati in due classi.

- Algoritmi che possono sbagliare, cioè dare una risposta errata[2]. Lo svantaggio qui è evidente.
- Algoritmi che non sbagliano; lo svantaggio è che l'algorimo può non essere così efficiente come ci si aspetta e che addirittura in alcuni casi potrebbe non fornire la risposta continuando l'esecuzione all'infinito (se la fornisce essa è corretta)[3].

[2] Questo tipo di algoritmo randomizzato viene chiamato di tipo *Monte Carlo*.

[3] Questo tipo di algoritmo randomizzato viene chiamato di tipo *Las Vegas*.

Ovviamente affinchè l'algoritmo sia utile è necessario che in entrambi i casi la probabilità con cui lo svantaggio si manifesta deve essere sufficientemente bassa.

L'uso della casualità negli algoritmi richiede nozioni di probabilità. In alcuni casi gli algoritmi randomizzati sono basati effettivamente su conoscenze di argomenti probabilistici molto complessi. Tuttavia spesso è sufficiente un minimo di tali nozioni per poter comprendere molti algoritmi randomizzati, alcuni anche di fondamentale importanza.

4.2 Randomness

Prima di procedere con lo studio degli algoritmi randomizzati, facciamo una breve, informale riflessione sui valori casuali di cui gli algoritmi hanno bisogno. Cosa è un valore casuale? Un valore casuale è un valore che non è predicibile. Supponiamo di considerare singoli bit (se ci servono numeri più grandi possiamo usare sequenze di bit). Un bit b è casuale se a priori non possiamo dire nulla sul suo valore e nel momento in cui lo dobbiamo usare il bit b varrà 0 con probabilità $1/2$ e ovviamente 1 con la stessa probabilità. Più in generale una variabile X che può assumere n valori è casuale se a priori non possiamo dire nulla sul suo valore e nel momento in cui la dobbiamo usare ognuno degli n possibili valori ha la stessa probabilità, $1/n$, di essere quello assunto dalla variabile In altre parole la distribuzione di probabilità sui possibili valori

deve essere quella uniforme[4]. I computer sono dotati di generatori di valori casuali, ma tali generatori non sono veramente casuali! La cosa non dovrebbe sorprendere perchè, per fortuna, tutto ciò che un computer fa è deterministico, quindi non c'è spazio per l'imponderabile. I generatori presenti sui computer *simulano*, in vari modi, la generazione di numeri casuali. In molti casi questa simulazione fornisce delle proprietà sufficienti (i numeri sembrano abbastanza casuali e, ad esempio, possono essere usati per implementare un videogioco) in altri casi le proprietà garantite non sono sufficienti (ad esempio per applicazioni crittografiche per le quali è fondamentale avere dei numeri veramente casuali).

[4] Più in generale una variabile casuale non deve necessariamente avere un distribuzione uniforme. Per i nostri scopi considereremo solo variabili casuali uniformemente distribuite.

Il problema della generazione di numeri casuali è un problema di estrema importanza; il lettore potrà approfondire questo aspetto su altri testi. Per i nostri scopi assumeremo di avere a disposizione un generatore di numeri veramente casuali. Più formalmente assumeremo che gli algoritmi randomizzati abbiano accesso ad una funzione $\texttt{random}(n)$ che restituisce un valore casuale uniformemente distribuito fra 1 e n. Gli algoritmi potranno sfruttare questa funzione dalla quale potranno attingere numeri casuali. Nello pseudocodice utilizzeremo la notazione

$$x \leftarrow \texttt{random}(n)$$

per indicare che stiamo utilizzando il generatore di numeri casuali che assegna ad x un valore casuale uniformemente distribuito fra 1 e n. Per comodità estenderemo tale notiazione ad insiemi: dato un insieme S di n elementi la notazione

$$x \leftarrow \texttt{random}(S)$$

indicherà la scelta casuale di uno degli elementi di S. Questo equivale a scgeliere un valore casuale $i = \texttt{random}(n)$ e poi selezionare l'i-esimo elemento dell'insieme.

4.3 Cenni di probabilità

Lemma 4.3.1 *Consideriamo un esperimento casuale in cui la probabilità di successo dell'esperimento è p. Se ripetiamo l'esperimento ed ogni ripetizione è indipendente dalle precedenti, allora il numero atteso di ripetizione per avere il primo successo è $1/p$.*

4.4 Problemi di ordinamento

Iniziamo con due problemi affini: l'ordinamento di un insieme e la ricerca del k-esimo elemento di un insieme non ordinato. Il secondo di questi problemi può essere facilmente ridotto al primo: per trovare il k-esimo elemento di un insieme non ordinato basta ordinarlo e prendere l'elemento nella k-esima posizione.

4.4.1 QuickSort randomizzato

Iniziamo dal problema dell'ordinamento e consideriamo l'algoritmo QuickSort. Esso prende in input una sequenza di elementi $a_1, a_2, \ldots, a_n$ sui quali è definita una relazione di ordine totale $\leq$ e fornisce in output gli elementi ordinati, cioè trova una permutazione

$a'_1, a'_2, \ldots, a'_n$ degli elementi tali che $a'_i \leq a'_j$ per $i < j$. Per semplicità assumeremo che tutti gli elementi siano distinti: modificare l'algoritmo per gestire anche elementi uguali è semplice e non richiede nessuna nuova idea ma solo delle accortezze non difficili da implementare. QuickSort opera nel seguente modo: seleziona uno specifico elemento della sequenza di input; poichè l'algoritmo funziona qualunque sia l'elemento scelto, solitamente si sceglie il primo elemento a_1. Tale elemento, detto *perno*, viene utilizzato per suddividere la sequenza di input in due parti $S^<$ e $S^>$, la prima contenente tutti gli elementi più piccoli e la seconda tutti gli elementi più grandi lasciando fuori dalla divisione il perno. Quindi QuickSort potrà essere usato ricorsivamente[5] su $S^<$ e $S^>$ per ottenere l'insieme ordinato che è dato dall'output relativo a $S^<$, seguito dal perno a_1, seguito dall'output relativo a $S^>$.

[5] Volendo considerare la possibilità di elementi uguali, essi possono essere trattati come il perno, cioè esclusi dalla ricorsione e poi dati in output insieme al perno.

La complessità di QuickSort è data dalla seguente relazione di ricorrenza:

$$T(n) = T(|S^<|) + T(|S^>|) + O(n),$$

dove il termine $O(n)$ è quello necessario ad effettuare la partizione. La soluzione di questa relazione di ricorrenza dipende da come viene diviso l'insieme di input nei due sottoinsiemi. Il caso migliore si ha quando i due sottoinsiemi $S^<$ e $S^>$ hanno la stessa grandezza, circa $n/2$, e quindi la soluzione della ricorrenza è $\Theta(n \log n)$. Il caso peggiore, che misura la complessità dell'algoritmo, si ha invece quando i due sottoinsiemi sono molto sbilanciati, ad esempio uno è vuoto e l'altro contiene $n-1$ elementi. In questo caso la soluzione della ricorrenza è $\Theta(n^2)$. Quindi la complessità di QuickSort è $O(n^2)$. É facile costruire sequenze di input che causano il caso pessimo: un input già ordinato (o quasi) sia in senso ascendente che discendente. Pertanto un "avversario" in grado di decidere l'input può causare sempre il caso pessimo.

Un approccio randomizzato può migliorare QuickSort. L'approccio randomizzato si basa sul fatto che l'algoritmo funziona qualunque sia il perno scelto. Nella versione deterministica si deve necessariamente scegliere una determinata posizione ed usare l'elemento in quella posizione come perno. Questo dà la possibilità a chi sceglie l'input di sceglierlo in modo tale che il perno sia sempre il minimo o il massimo (o quasi) dell'insieme causando una partizione complemetamente (o quasi completamente) sbilanciata. Poichè, come detto, l'algoritmo funziona qualunque sia il perno, avendo a disposizione un generatore di numeri casuali, anzichè fissare la posizione del perno, cosa necessaria per un algoritmo deterministico, possiamo sceglierlo in maniera casuale selezionandolo durante l'esecuzione dell'algoritmo. Pertanto la versione randomizzata differisce da quella deterministica solo per il fatto che il perno viene scelto in maniera casuale.

Algorithm 16: RandomQuickSort

```
RandomQuickSort(S)
if |S| sufficientemente piccolo, per esempio 1,2,3 then
    Ordina S con forza bruta
    Fornisci in output gli elementi di S
else
    i ← random(n)
    /* a_i è il perno scelto in modo casuale                    */
    for ogni a_j ∈ S do
        Se a_j < a_i allora S^< = S^< ∪ {a_j}
        Se a_j > a_i allora S^> = S^> ∪ {a_j}
    Chiama ricorsivamente RandomQuickSort(S^<)
    Chiama ricorsivamente RandomQuickSort(S^>)
    Fornisci in output gli elementi di S^<, seguiti da a_i, seguito dagli elementi di
     S^>.
```

Poichè RandomQuickSort differisce da QuickSort solo per la scelta del perno, la relazione di ricorrenza che descrive la complessità dell'algoritmo è la stessa. Quindi sia la complessità del caso migliore che quella del caso peggiore sono le stesse.

Cosa abbiamo guadagnato quindi? La differenza fra i due algoritmi è che per RandomQuickSort il caso pessimo non può più essere determinato dall'input ma può scaturire solo dalla scelte casuali del perno. In altre parole per avere il caso pessimo dobbiamo essere veramente sfortunati (è in gioco la casualità!). Poichè le scelte del perno sono casuali la complessità attesa di RandomQuickSort è quella del caso medio, che risulta essere uguale al caso ottimo, cioè $O(n \log n)$.

L'analisi formale non è semplicissima, pertanto procediamo con delle osservazioni che ci forniscono un'idea sul comportamento atteso dell'algoritmo. Successivamente procederemo con l'analisi formale.

Prima di tutto osserviamo che il tempo di cui necessita l'algoritmo, escludendo le chiamate ricorsive, è praticamente quello necessario per operare la partizione in quanto tutte le altre operazioni possono essere eseguite in tempo costante e che il tempo necessario per la partizione è lineare nel numero n di elementi da considerare. Non dovrebbe essere difficile convincersi che il tempo necessario è proporzionale al numero di confronti da fare che è esattamente $n - 1$ (per operare la partizione è sufficiente confrontare ogni elemento con il perno). Quindi, ignorando il -1, stimeremo con cn il tempo necessaro ad effettuare la partizione, per una qualche costante c.

Se siamo veramente fortunati, il perno divide sempre l'insieme di input in due parti uguali, generando la relazione di ricorrenza

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

la cui soluzione è $O(n \log n)$, mentre se siamo veramente sfortunati la partizione produce un insieme vuoto e uno di $n - 1$ elementi, generando la relazione di ricorrenza

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

la cui soluzione è $O(n^2)$. Questi sono i due casi estremi che corrispondono al caso migliore e al caso pessimo. Cosa succede in mezzo? Supponiamo ad esempio che la partizione generi sempre due insiemi leggermente sbilanciati, ad esempio con $1/4$ e $3/4$ degli elementi. La relazione di ricorrenza diventa

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

La soluzione di questa relazione è ancora $O(n \log n)$. Proviamo a sbilanciare maggiormente i due insiemi della ricorsione, ad esempio $1/10$ e $9/10$. La relazione di ricorrenza diventa

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn.$$

Nonostante questo sbilanciamento più pronunciato, la soluzione è ancora $O(n \log n)$. In realtà fino a che lo sbilanciamento prevede una frazione (funzione di n) da una parte e il resto dall'altro, per quanto piccola possa essere tale frazione, la soluzione alla relazione di ricorrenza sarà sempre $O(n \log n)$.

La soluzione diventerà quadratica solo quando la partizione inserirà in uno dei due sottoinsiemi un numero di elementi più piccolo di *una costante*. Ad esempio se la partizione produce sempre un insieme di 10 elementi ed un insieme di $n - 10$ elementi, allora la relazione di ricorrenza diventa

$$T(n) = T(10) + T(n - 10) + cn$$

la cui soluzione è $O(n^2)$. Chiaramente anche in questo caso, simmetricamente a quanto detto prima, partizioni leggermente più bilanciate, ad esempio con 100 e $n - 100$ elementi o 3000 e $n - 3000$ elementi, generano sempre una relazione di ricorrenza la cui soluzione è $O(n^2)$.

Un altro caso interessante da analizzare è il seguente. Supponiamo che la fortuna ci sia amica a corrente alternata: ad una iterazione siamo fortunati con una partizione completamente bilanciata mentre alla successiva siamo sfortunati con una partizione completamente sbilanciata e così via, alternando fortuna a sfortuna. Cosa succede al tempo di esecuzione? Sostituendo $T(n)$ con $F(n)$ e $S(n)$, rispettivamente per il caso fortunato e quello sfortunato, avremo che

$$F(n) = 2S(n/2) + cn,$$

$$S(n) = F(n - 1) + cn.$$

Quindi si ha che

$$\begin{aligned} F(n) &= 2S(n/2) + cn \\ &= 2\left(F\left(\frac{n}{2} - 1\right) + cn/2\right) + cn \\ &= 2F\left(\frac{n}{2} - 1\right) + cn + cn \\ &= O(n \log n) \end{aligned}$$

e in modo simile si può vedere che $S(n) = O(n \log n)$.

Dunque l'intuizione ci dice che nella maggior parte dei casi il tempo di esecuzione di RandomQuickSort è dato da una relazione di ricorrenza la cui soluzione è $O(n \log n)$ per cui non dovrebbe sorprendere il fatto che la complessità (attesa) di RandomQuickSort sia $O(n \log n)$. Vediamo questo fatto in maniera più formale.

Sia $T(n)$ una variabile casuale che descrive il tempo di esecuzione di RandomQuickSort su un input di n elementi. Per $k = 0, 1, \ldots, n-1$, sia X_k una variabile (casuale) indicatrice definita da

$$X_k = \begin{cases} 1 \text{ se la partizione è } k : n-k-1 \\ 0 \text{ altrimenti.} \end{cases}$$

Poichè il perno è scelto uniformemente a caso, tutti le possibili partizioni $k : n-k-1$ hanno la stessa probabilità di verificarsi, e qiondi si ha che

$$E[X_k] = \Pr\{X_k = 1\} = 1/n.$$

Quindi si ha che

$$T(n) = \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + n).$$

Pertanto

$$\begin{aligned} E[T(n)] &= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + n)] \\ &\quad \textit{(per la linearità di E)} \\ &= \sum_{k=0}^{n-1} E[X_k] \cdot E[(T(k) + T(n-k-1) + n)] \\ &\quad \textit{(per l'indipendenza delle scelte casuali)} \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E[(T(k) + T(n-k-1) + n)] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} n \\ &= n + \frac{2}{n} \sum_{k=0}^{n-1} E[(T(k)] \\ &\quad \textit{(perchè le due sommatorie sono uguali)} \end{aligned}$$

A questo punto abbiamo una relazione di ricorrenza per $E[T(n)]$ che possiamo risolvere ad esempio con il metodo della sostituzione. Con tale metodo dobbiamo provare a "indovinare" una soluzione e verificare se la scelta è corretta. Per quanto detto prima possiamo ragionevolmente tentare con $E[T(n)] \leq an \log n$, per una qualche costante a.

$$
\begin{aligned}
E[T(n)] &= n + \frac{2}{n}\sum_{k=0}^{n-1} E[(T(k)] \\
&\leq n + \frac{2}{n}\sum_{k=0}^{n-1} ak\log k \\
&= n + \frac{2a}{n}\sum_{k=1}^{n-1} k\log k \\
&\leq n + \frac{2a}{n}\int_{x=1}^{n} x\log x \\
&\quad \left(\text{ricordando che } \int x\log x = \frac{x^2\log x}{2} - \frac{x^2}{4}\right) \\
&= n + \frac{2a}{n}\left[\frac{x^2\log x}{2} - \frac{x^2}{4}\right]_1^n \\
&\leq n + \frac{2a}{n}\left(\frac{n^2\log n}{2} - \frac{n^2}{4} + \frac{1}{4}\right) \\
&= n + an\log n - \frac{an}{2} + \frac{a}{2n}.
\end{aligned}
$$

A questo punto basta scegliere la costante a in modo tale che la relazione sia verificata a partire da un determinato valore di n in poi. Scgliendo $a = 2$ la relazione è sempre verificata, per ogni $n \geq 1$.

Una versione alternativa di RandomQuickSort, è quella in cui il perno viene forzato ad essere un elemento che fa ottenere sempre un buon bilanciamento. L'algoritmo è equivalente, richiede qualche piccolo sforzo in più per la sua implementazione, ma è più facile da analizzare. Pertanto è interessante presentarlo ed analizzarlo. Diremo che un perno è *centrale* se ci sono almeno un quarto degli elementi che sono più piccoli del perno ed almeno un quarto degli elementi che sono più grandi del perno, cioè entrambi $S^<$ e $S^>$ hanno cardinalità almeno $n/4$. In RandomQuickSortBil forzeremo il perno ad essere un elemento centrale.

Algorithm 17: RandomQuickSortBil

```
RandomQuickSortBil(S)
if |S| sufficientemente piccolo, per esempio 1,2,3 then
    Ordina S con forza bruta
    Fornisci in output gli elementi di S
else
    while Non troviamo un perno centrale do
        i ← random(n)
        /* a_i è il perno scelto in modo casuale                    */
        for ogni a_j ∈ S do
            Se a_j < a_i allora S^< = S^< ∪ {a_j}
            Se a_j > a_i allora S^> = S^> ∪ {a_j}
        if |S^<| ≥ |S|/4 e |S^>| ≥ |S|/4 then
            abbiamo trovato un perno centrale (a_i)
    Chiama ricorsivamente RandomQuickSortBil(S^<)
    Chiama ricorsivamente RandomQuickSortBil(S^>)
    Fornisci in output gli elementi di S^<, seguiti da a_i, seguito dagli elementi di
     S^>.
```

Con RandomQuickSortBil siamo sempre "fortunati" nella ricorsione in quanto il caso pessimo che si verifica è uno sbilanciamento di $n/4 : 3n/4$. Come abbiamo già visto questo sbilanciamento genera comunque una ricorrenza la cui soluzione è $O(n \log n)$. Tuttavia questa partizione fortunata viene pagata con il ciclo **while** che cerca un perno centrale: la ricorsione verrà fatta solo dopo aver trovato un perno centrale, il che significa che non sappiamo quante partizioni dobbiamo provare prima di trovarne una bilanciata. Tuttavia possiamo facilmente stimare il numero medio di tentativi che dobbiamo fare per trovare un perno centrale. Infatti, poichè il perno è scelto uniformemente a caso, si ha che la probabilità di trovare un perno centrale è di $1/2$ in quanto ci sono esattamente $n/2$ elementi che sono centrali (sono gli elementi dalla posizione $\frac{n}{4}$ alla posizione $\frac{3}{4}n$ nella sequenza ordinata).

Quindi per il Lemma 4.3.1 si ha che il numero atteso di iterazioni nel ciclo **while** prima di trovare un elemento centrale è 2. Questo significa che il tempo necessario al partizionamento, in media, raddoppia. Pertanto la relazione di ricorrenza del tempo di esecuzione atteso di RandomQuickSortBil è

$$E[T(n)] = E[T(n/4)] + E[T(3n/4)] + 2cn,$$

e pertanto $E[T(n)] = O(n \log n)$.

4.4.2 *k-esimo ordine statistico*

Consideriamo adesso un problema collegato a quello dell'ordinamento: la ricerca del k-esimo ordine statistico. Dato un insieme di n elementi $S = \{a_1, a_2, \ldots, a_n\}$, il k-esimo ordine statistico è l'elemento che si troverrebbe nella posizione k se gli elementi fossero

elencati in ordine crescente. Come per il caso di QUICKSORT, supporremo che gli a_i siano tutti diversi tra loro; anche in questo caso la presenza di elementi uguali non è un problema, ma assumendo che gli elementi siano tutti diversi semplifichiamo l'analisi; le idee fondamentali che introdurremo per risolvere il problema continuano ad essere valide anche se i numeri non sono tutti diversi.

Il *minimo* dell'insieme S è il primo ordine statistico, mentre il *massimo* è l'ultimo (n-esimo) ordine statistico. La *mediana* dell'insieme S è l'ordine statistico che si trova al centro della lista ordinata. Per n dispari, la mediana è il k-esimo ordine statistico con $k = (n+1)/2$. Un valore di n pari crea un problema per la definizione di centro in quanto, tolta la mediana, i due insiemi non possono avere la stessa cardinalità. Quindi uno dei due insieme dovrà avere un elemento in più e questo da spazio a due possibili mediane, cioè il k-esimo ordine statistico $k = n/2$ oppure con $k = (n/2) + 1$; una qualsiasi delle due definizioni va bene.

Dovrebbe essere ovvio che è facile trovare il k-esimo ordine statistico in $O(n \log n)$: basta ordinare S e prendere l'elemento in posizione k. La domanda che ci poniamo è: è davvero necessario ordinare gli elementi per risolvere il problema? In altre parole è possibile progettare un algoritmo che trova il k-esimo ordine statistico in meno di $O(n \log n)$? Vedremo un algoritmo randomizzato[6] che risolve il problema in $O(n)$; ovviamente tale tempo è quello atteso, le scelte casuali potrebbero comunque causare il caso pessimo che richiede $O(n \log n)$. L'algoritmo RANDOMSELECT è un algoritmo randomizzato che risolve il problema del k-esimo ordine statistico. RANDOMSELECT$(S, 1)$ corrisponde al minimo, RANDOMSELECT(S, n) al massimo e la mediana è data da RANDOMSELECT$(S, n/2)$ per n pari o da RANDOMSELECT$(S, (n+1)/2)$ per n dispari.

[6] Esiste anche un algoritmo deterministico che risolve il problema in $O(n)$. Tuttavia quello randomizzato è molto più semplice.

La struttura di base dell'algoritmo RANDOMSELECT è la seguente. Scegliamo un elemento $a_i \in S$ che chiameremo perno; la scelta del perno è fatta in modo casuale. Quindi partizioneremo i restanti elementi dell'insieme S in due sottoinsiemi $S^< = \{a_j | a_j < a_i\}$ e $S^> = \{a_j | a_j > a_i\}$. In base alle cardinalità di tali insiemi potremo determinare quale dei due contiene il k-esimo elemento (che potrebbe anche essere proprio a_i) e poi procedere ricorsivamente se non lo abbiamo ancora trovato.

Algorithm 18: RANDOMSELECT

```
RandomSelect(S, k)
i ← random(n)
/* a_i è il perno scelto in modo casuale                    */
for ogni a_j ∈ S do
    Se a_j < a_i allora S^< = S^< ∪ {a_j}
    Se a_j > a_i allora S^> = S^> ∪ {a_j}
if |S^<| = k − 1 then
    Il perno a_i è il valore cercato
if |S^<| ≥ k − 1 then
    /* L'elemento cercato si trova in S^<                   */
    Chiama ricorsivamente RandomSelect(S^<, k)
if |S^<| < k − 1 then
    /* L'elemento cercato si trova in S^>                   */
    ℓ = |S^<|
    Chiama ricorsivamente RandomSelect(S^>, k − 1 − ℓ)
```

Lemma 4.4.1 *L'algoritmo* RANDOMSELECT *restituisce il k-esimo ordine statistico.*

DIMOSTRAZIONE. Si noti che se $|S| = 1$ allora si deve avere $k = 1$ e l'algoritmo restituisce l'unico elemento di S. Se $|S| > 1$, l'algoritmo ricorsivamente cerca l'elemento in un insieme più piccolo aggiustando oppurtunamente il valore di k. Il fatto che il perno venga scelto in modo casuale non ha nessuna influenza sulla correttezza dell'algoritmo. Pertanto l'algoritmo termina e restituisce la risposta corretta. □

Quale è il tempo di esecuzione di RANDOMSELECT? Ovviamente dipende da quanto sono i grandi i sottoproblemi delle chiamate ricorsive. Se ad esempio il perno fosse sempre la mediana dell'insieme su cui si effettua la chiamata ricorsiva allora si avrebbe $T(n) = T(n/2) + cn$. La soluzione di questa relazione di ricorrenza è $T(n) = O(n)$.

Se invece il perno fosse sempre il minimo (o il massimo), allora la relazione di ricorrenza sarebbe $T(n) = T(n-1) + cn$, la cui soluzione è $T(n) = \Theta(n^2)$.

Chiaramente è difficile scegliere la mediana come perno visto che uno degli obiettivi di RANDOMSELECT è proprio quello di trovare la mediana. Tuttavia anche un elemento che è vicino al centro comporta una relazione di ricorrenza la cui soluzione è $O(n)$. Infatti supponiamo che la grandezza dell'insieme su cui si effettua la chiamata ricorsiva sia tale da garantire che ci siano almeno ϵn elementi sia più piccoli che più grandi del perno, per una qualsiasi costante $\epsilon > 0$. Allora si ha che l'insieme su cui si chiama ricorsivamente l'algoritmo non può avere più di $(1-\epsilon)n$ elementi. Quindi il tempo di esecuzione soddisfa $T(n) \leq T((1-\epsilon)n) + cn$. Ricordando la serie geometrica $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, si ha che la soluzione di questa ricorrenza è

$$\begin{aligned} T(n) &\leq T((1-\epsilon)n) + cn \\ &\leq T((1-\epsilon)^2 n) + c(1-\epsilon) + cn \\ &\leq \dots \end{aligned}$$

$$\begin{aligned} &= cn\left(1 + (1-\epsilon) + (1-\epsilon)^2 + \ldots\right) \\ &\leq \tfrac{1}{\epsilon} cn. \end{aligned}$$

Quindi, i casi che generano tempi di esecuzione non lineari sono quelli in cui il perno è estremamente vicino al minimo o al massimo. Questo suggerisce che la scelta di un perno casuale nella maggior parte dei casi funziona bene.

Per analizzare l'algoritmo in modo più formale utilizzeremo la seguente terminologia: diremo che l'algoritmo è nella *fase* j, quando il numero di elementi dell'insieme in considerazione, che si riduce ad ogni chiamata ricorsiva, è maggiore di $n(\frac{3}{4})^{j+1}$ ma minore o uguale a $n(\frac{3}{4})^j$. Per valutare la complessità dell'algoritmo daremo un limite al tempo speso nella fase j. Inoltre diremo che un elemento è *centrale* rispetto all'insieme in considerazione, se ci sono almeno un quarto di elementi più piccoli ed almeno un quarto di elementi più grandi.

Dunque, se il perno è un elemento centrale, nella successiva chiamata ricorsiva il numero di elementi in considerazione diminuirà di almeno un quarto e pertanto la fase corrente terminerà. La probabilità che il perno sia centrale è $1/2$ in quanto metà degli elementi sono centrali. Quindi per il Lemma 4.3.1 si ha che il numero atteso di iterazioni prima di scegliere un perno centrale è 2. Quindi in ogni fase j il numero atteso di iterazioni eseguite nella fase j è 2.

Per concludere l'analisi, denotiamo con X la variabile casuale che descrive il numero di passi eseguiti dall'algoritmo, e denotiamo con X_j il numero di passi eseguiti dall'algoritmo nella fase j. Quindi $X = X_0 + X_1 + X_2 + \ldots$. Quando l'algoritmo si trova nella fase j l'insieme preso in considerazione contiene al massimo $n(\frac{3}{4})^j$ elementi, poichè il lavoro svolto dall'algoritmo in una singola iterazione è proporzionale a tale numero, si ha che per ogni iterazione durante la fase j, l'algoritmo spende al massimo tempo $cn(\frac{3}{4})^j$, per una costante c. Sappiamo già che il numero atteso di iterazioni per ogni fase è 2 quindi si ha che $E[X_j] \leq 2cn(\frac{3}{4})^j$. Per la linearità della media, si ha che $E[X] = \sum_j E[X_j]$ e pertanto

$$E[X] = \sum_j E[X_j] \leq \sum_j 2cn\left(\frac{3}{4}\right)^j = 2cn\sum_j\left(\frac{3}{4}\right)^j \leq 8cn = O(n).$$

4.5 *Taglio minimo globale*

Dato un grafo non direzionato $G = (V, E)$, definiamo un *taglio* di G una partizione di V in due insiemi non vuoti A e B. Per un taglio (A, B) di G, la grandezza di (A, B) è il numero di archi che attraversano il taglio, cioè che hanno un vertice in A e l'altro in B. Un taglio minimo globale è un taglio di grandezza minima.

L'aggettivo globale sta ad enfatizzare il fatto che il taglio può essere qualunque partizione di V in due sottoinsiemi, non c'è nessuna sorgente e nessun pozzo, come invece avviene per i problemi di flusso. L'unica restrizione è che gli insiemi A e B non devono essere vuoti, altrimenti avremmo un taglio di grandezza 0 ed il problema non avrebbe senso.

Il taglio minimo globale può essere visto come un parametro di *robustezza* del grafo.

Infatti è il più piccolo numero di archi che si deve cancellare per disconnettere il grafo stesso.

Ovviamente il problema è in qualche modo legato ai problemi di flusso per i quali sappiamo che il taglio minimo gioca un ruolo fondamentale; il teorema del flusso massimo e taglio minimo ci dice che il massimo flusso che può arrivare dalla sorgente alla destinazione è uguale alla somma delle capacità del taglio minimo che separa la sorgente dalla destinazione. Quindi non dovrebbe sorprendere il fatto che possiamo usare l'algoritmo per il calcolo del massimo flusso, e quindi del taglio minimo, per risolvere anche il problema del taglio minimo globale.

Lemma 4.5.1 *Esiste un algoritmo polinomiale per trovare un taglio minimo globale di un grafo non direzionato G.*

Dimostrazione. Sia $G = (V, E)$ il grafo. Possiamo sfruttare l'algoritmo per il calcolo del massimo flusso che di fatto trova anche il taglio minimo. Dobbiamo però prestare attenzione a due differenze fra i problemi.

La prima è dovuta al fatto che il massimo flusso gestisce delle capacità su degli archi direzionati. Questo è una caratteristica in più che possiamo facilmente gestire e sfruttare. Creiamo un nuovo grafo $G' = (V', E')$ con $v' = v$ e per ogni arco $(u, v) \in E$, inseriamo in E' sia l'arco (u, v) che l'arco (v, u) e ad entrambi assegniamo capacità 1. Questo fa sì che il valore del flusso massimo sia proprio il numero di archi che attraversano il taglio e quindi la grandezza del taglio nella definizione del problema del taglio minimo globale.

La seconda differenza è che abbiamo bisogno di una sorgente e di un pozzo per poter calcolare il flusso massimo. Supponiamo di scegliere una qualsiasi coppia di nodi a, b come sorgente e pozzo. A questo punto usando l'algoritmo per il calcolo del massimo flusso individueremo il taglio minimo fra tutti i tagli che separano a da b. Chiaramente questo potrebbe non essere il taglio minimo globale, per il quale magari a e b sono nello stesso insieme. Tuttavia il nodo a deve necessariamente essere separato da un qualche altro nodo! Quindi se ripetiamo l'esecuzione dell'algoritmo per $n - 1$ volte, usando ogni volta a come sorgente e tutti gli altri $n - 1$ nodi come pozzo, individueremo il taglio minimo globale che è semplicimente il taglio minimo fra tutti gli $n - 1$ trovati nelle ripetizioni dell'algoritmo di massimo flusso.

La complessità totale è data da $n - 1 = O(n)$ moltiplicato la complessità dell'algoritmo di massimo flusso. Essendo quest'ultima polinomiale, anche la complessità del calcolo del taglio minimo globale è polinomiale. □

Quanto appena detto farebbe supporre che il calcolo del taglio minimo globale è più difficile del problema del massimo flusso in quanto per risolvere il primo dobbiamo risolvere $n - 1$ istanze del secondo. Non è così, nel senso che si può risolvere il problema del taglio minimo globale in maniera più efficiente sfruttando la randomizzazione.

4.5.1 Algoritmo di contrazione degli archi

Descriveremo l'algoritmo nella sua forma più semplice. Questa forma, sebbene sia comunque polinomiale, non è quella più efficiente; successive ottimizzazioni hanno

reso l'algoritmo più efficiente. Tuttavia questa forma ci permette di capire facilmente l'idea di base.

L'algoritmo di contrazione opera collassando i nodi: due nodi vengono fusi insieme e diventano un solo nodo. Questo crea archi multipli fra nodi collassati insieme. Quindi per gestire questa possibilità considereremo il grafo $G = (V, E)$ come un *multigrafo*: in un multigrafo è possibile avere più di un arco fra due nodi. Quindi l'insieme E è un multi-insieme in cui un elemento (u, v) può comparire più di una volta. Inizialmente il grafo G è il grafo di input, quindi non è un multigrafo. Durante l'esecuzione dell'algoritmo però i nodi verrano collassati e questo potrà creare un multigrafo.

Ad esempio, consideramo la Figura 4.3. Il grafo iniziale è un grafo normale con 4 nodi uniti da alcuni archi. Collassando il nodo a con il nodo b, poichè sia a che b avevano un arco con d, nel nuovo grafo il nodo $\{a, b\}$ avrà due archi che lo collegano a d.

Se da questo nuovo grafo si collassa il nodo $\{a, b\}$ con il nodo d creando il nodo $\{a, b, d\}$, questo nodo avrà due archi con il nodo c. Anche in questo caso archi che per effetto del collassamento di due nodi diventerebbero degli archi che collegano un nodo a sè stesso sono stati rimossi. Ad esempio, nel collassare a e b l'arco (a, b) viene rimosso; nel collassare $\{a, b\}$ con d i due archi $(\{a, b\}, d)$ vengono rimossi. I nodi creati nei collassamenti verranno chiamati *supernodi*. Un supernodo w può essere identificato con l'insieme dei nodi $S(w)$ che sono stati collassati per ottenerlo.

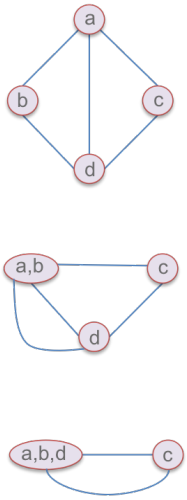

Figura 4.3: Contrazione di nodi e multigrafo

L'algoritmo di contrazione procede collassando nodi su archi scelti a caso fino a che il multigrafo contiene due soli nodi v_1 e v_2. A questo punto l'algoritmo termina e produce come output i due insieme $S(v_1)$ e $S(v_2)$.

Algorithm 19: EdgeContraction$(G = (V, E))$

```
Per ogni nodo v, S(v) = {v}
if V ha due soli nodi v1 e v2 then
    Fornisci in output S(v1) e S(v2)
else
    (u, v) ← random(E)
    Sia G' il grafo ottenuto collassando u e v nel nodo z
    S(z) = S(u) ∪ S(v)
    EdgeContraction(G')
```

Analizziamo l'algoritmo. È possibile implementare l'algoritmo in tempo $O(n^2)$. Tuttavia una singola esecuzione dell'algoritmo non sarà sufficiente in quanto la probabilità di trovare una taglio minimo è bassa ma sufficientemente alta da poter essere sfruttata ripetendo varie volte l'algoritmo. Quanto è grande questa probabilità? Vediamo.

Prima di tutto osserviamo che che EdgeContraction restituisce un taglio. Infatti i due supernodi finali rappresentano due sottoinsiemi di V che sono una partizione di V visto che tutti i nodi sono stati collassati o nell'uno o nell'altro. Tale taglio è il risultato di scelte casuali, dipende infatti dalla sequenza di archi scelti per collassare i nodi. Quindi è possibile, se siamo fortunati, che il taglio fornito dall'algoritmo sia proprio un taglio minimo. Ovviamente è anche possibile, e più probabile, che venga

restituito un taglio non minimale. Ma quale è la probabilità di ognuno di queste due possibilità? Si potrebbe pensare che la probabilità che il taglio restituito sia minimo è estremamente piccola. Ovviamente non è alta, ma è sufficientemente non piccola (è solo polinomialmente piccola) da poter ripetere l'algoritmo un numero polinomiale di volte e scegliendo il taglio più piccolo fra tutti quelli trovati riusciamo a trovare un taglio minimo globale con alta probabilità.

Lemma 4.5.2 *L'algoritmo* EdgeContraction *produce un taglio minimo globale con probabilità almeno* $1/\binom{n}{2}$.

Dimostrazione. Sia (A, B) un taglio minimo globale e sia k la sua grandezza. In altre parole l'insieme di archi con un vertice in A ed un vertice in B, che denoteremo con F, ha cardinalità k.

Sia (A', B') il taglio restituito dall'algoritmo. Vogliamo provare un limite inferiore di $1/\binom{n}{2}$ alla probabilità che $(A', B') = (A, B)$.

L'algoritmo non troverà il taglio (A, B) se un arco di F viene selezionato per collassare i suoi due nodi. In tale caso entrambi i nodi finiranno nello stesso supernodo e quindi saranno o entrambi in A' oppure entrambi in B'. D'altra parte se nessun arco di F viene selezionato durante l'algoritmo allora si ha che $A' = A$ e $B' = B$.

Poichè la grandezza del taglio minimo globale è k, ogni nodo ha almeno k vertici ad esso incidenti. Infatti se così non fosse, dato un nodo v con meno di k archi incidenti, il taglio $(\{v\}, V \setminus \{v\})$ sarebbe un taglio con grandezza minore del taglio minimo, che è un assurdo. Quindi possiamo dedurre che il numero di archi nel grafo iniziale è $|E| \geq \frac{1}{2}nk$.

Definiamo β_i come il seguente evento: un arco di F viene contratto alla i-esima iterazione. L'evento α_i è definito come il complemento di β_i: nessun arco di F viene contratto alla i-esima iterazione dell'algoritmo. Ovviamente si ha che $Pr[\alpha_i] = 1 - Pr[\beta_i]$.

Dunque si ha

$$Pr[\beta_1] \leq \frac{k}{\frac{1}{2}nk} = \frac{2}{n}$$

e pertanto

$$Pr[\alpha_1] \geq 1 - \frac{2}{n}.$$

Per la nostra analisi servirà valutare le probabilità

$$Pr[\alpha_{j+1} | \alpha_1 \cap \alpha_2 \ldots \cap \alpha_j].$$

Consideriamo $Pr[\alpha_2|\alpha_1]$. La condizione α_1 significa che nella prima iterazione non è stato selezionato un arco di F, quindi gli archi di F sono ancora k, mentre i nodi del grafo sono $n - 1$. Pertanto si ha che

$$Pr[\beta_2|\alpha_1] \leq \frac{k}{\frac{1}{2}k(n-1)} = \frac{2}{n-1}$$

e pertanto

$$Pr[\alpha_2|\alpha_1] \geq 1 - \frac{2}{n-1}.$$

Analogamente per le successive iterazioni. Più precisamente dopo ogni contrazione il numero di nodi del grafo diminuisce di un'unità. Quindi dopo j iterazioni, il grafo G' ha esattamente $n - j$ nodi e pertanto il numero di archi in G' è almeno $\frac{1}{2}k(n-j)$. Possiamo quindi generalizzare quanto detto poc'anzi sulla probabilità che un arco di F venga contratto. Quando si opera sul grafo G', all'iterazione $(j+1)$-esima, assumendo che nessun arco di F sia stato contratto nelle precedenti iterazioni, si ha che la probabilità che un arco di F venga contratto è al massimo

$$Pr[\beta_{j+1}|\alpha_1 \cap \alpha_2 \ldots \cap \alpha_j] \leq \frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}$$

e quindi si ha che

$$Pr[\alpha_{j+1}|\alpha_1 \cap \alpha_2 \ldots \cap \alpha_j] \geq 1 - \frac{2}{n-j}.$$

Come abbiamo già osservato, se durante le $n-2$ iterazioni dell'algoritmo nessun arco di F viene selezionato allora si ha che $A' = A$ e $B' = B$. Quindi ci interessa studiare la quantità

$$Pr[A' = A] = Pr[\alpha_1 \cap \alpha_2 \ldots \cap \alpha_{n-2}].$$

Sfruttando la formula per la probabilità condizionata[7] (usandola $n-2$ volte con $B = \alpha_i$ per $i = 1, \ldots n-2$) si ha che

[7] $Pr[A|B] = \frac{Pr[A \cap B]}{Pr[B]}$

$$\begin{aligned}
Pr[A' = A] &= Pr[\alpha_1 \cap \alpha_2 \ldots \cap \alpha_{n-2}] \\
&= Pr[\alpha_1] \cdot Pr[\alpha_2 \cap \alpha_3 \ldots \cap \alpha_{n-2}|\alpha_1] \\
&= Pr[\alpha_1] \cdot Pr[\alpha_2|\alpha_1] \cdot Pr[\alpha_3 \cap \ldots \cap \alpha_{n-2}|\alpha_1 \cap \alpha_2] \\
&= \ldots \\
&= Pr[\alpha_1] \cdot Pr[\alpha_2|\alpha_1] \cdot Pr[\alpha_3|\alpha_1 \cap \alpha_2] \cdot \ldots \cdot Pr[\alpha_{n-2}|\alpha_1 \cap \alpha_2 \ldots \cap \alpha_{n-3}] \\
&\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \ldots \left(1 - \frac{2}{n-j}\right) \ldots \left(1 - \frac{2}{3}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \ldots \left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)} \\
&= \binom{n}{2}^{-1}.
\end{aligned}$$

□

Il lemma precedente ci dice che in ogni singola esecuzione la probabilità che l'algoritmo EdgeContraction trovi un taglio minimo globale è almeno $\binom{n}{2}^{-1}$, o equivalentemente che la probabilità che l'algoritmo non trovi un taglio minimo globale è al massimo $1 - \binom{n}{2}^{-1}$.

Sebbene la probabilità di insuccesso con una singola esecuzione sia alta è sufficientemente più piccola di 1, e pertanto ripetendo l'algoritmo abbastanza volte, la probabilità che un taglio minimo non venga mai trovato si riduce a valori più interessanti.

Quando ripetiamo l'esecuzione dell'algoritmo per un numero x di volte, poichè ogni esecuzione è indipendente dalle altre, la probabilità di non trovare un taglio minimo in

nessuna delle x esecuzioni è

$$\left(1 - 1/\binom{n}{2}\right)^x.$$

Questa funzione tende a 0 al tendere di x a ∞, pertanto possiamo rendere piccola a piacere la probabilità di non trovare un taglio minimo. Ad esempio, se $n = 50$, la probabilità di trovare un taglio minimo con una singola esecuzione è almeno $2/2450 \simeq 0.0008$ e quindi la probabilità di non trovare un trovare un taglio minino in una singola esecuziine è al massimo 0.9992. Ripetendo l'algoritmo $n = 50$ volte, la probabilità di non trovare il taglio minimo è al massimo $0.9992^{50} =\simeq 0.96$; ripetendo l'algoritmo $\binom{n}{2} = 2450$ volte, la probabilità di non trovare il taglio minimo è al massimo $0.9998^{2450} =\simeq 0.14$; ripetendo l'algoritmo $\binom{n}{2} \log n \simeq 2450 \cdot 5.63 \simeq 13793$ volte, la probabilità di non trovare il taglio minimo è al massimo $0.9992^{13793} =\simeq 0.00001$, cioè lo 0.001%.

Tuttavia non possiamo esagerare con il numero di ripetizioni altrimenti l'intero processo diventa inefficiente. Il numero di ripetizioni dell'algoritmo deve essere al massimo polinomiale; $x = \binom{n}{2}$ e $x = \binom{n}{2} \log n$ sono polinomiali in n. Analizziamole più analiticamente.

Se $x = \binom{n}{2} = n(n-1)/2$ si ha che la probabilità di insuccesso è $\leq 1/e \simeq 0.37$. Infatti per un tale numero di ripetizioni la probabilità di insuccesso è

$$\left(1 - 1/\binom{n}{2}\right)^{\binom{n}{2}} \tag{4.1}$$

e la funzione $\left(1 - \frac{1}{x}\right)^x$ è una funzione crescente che converge da $1/4$ a $1/e \simeq 0,3678 < 0,37$ al crescere di x. Quindi la probabilità di trovare un taglio minimo è almeno $0,73$.

Per avere una probabilità di successo ancora più alta, possiamo usare $x = \binom{n}{2} \log n$, nel qual caso la probabilità di insuccesso è

$$\left(1 - 1/\binom{n}{2}\right)^{\binom{n}{2} \log n} \tag{4.2}$$

e tale funzione vale al massimo $e^{-\log n} = 1/n$. La Figura 4.4 mostra le funzioni (4.1), linea blu, e (4.2), linea rossa.

Per entrambe le scelte di x l'algoritmo è ancora polinomiale. Scegliendo $x = \binom{n}{2} \log n = O(n^2 \log n)$, poichè ogni singola esecuzione dell'algoritmo costa $O(n^2)$ il tempo totale è $O(n^4 \log n)$. Notiamo che rispetto all'algoritmo che risolve il problema usando un algoritmo per il massimo flusso, EdgeContraction ha un tempo di esecuzione peggiore. Infatti il massimo flusso può essere risolto in $O(n^3)$ e quindi ripentendolo $n - 1$ per risolvere il taglio minimo globale avremmo un tempo di esecuzione di $O(n^4)$.

Come abbiamo già detto, esistono versioni ottimizzate che hanno tempi di esecuzione considerevolmente migliori. Una variante semplice è quella di Karger-Stein che risolve il problema con un tempo di esecuzione di $O(n^2 \log n)$ che però fornisce una probabilità di successo più grande, $\Omega(1/\log n)$, per la quale servono solo $O(\log^2 n)$ ripetizioni per avere una ragionevole probabilità di successo ($\geq 1 - 1/poly(n)$). Quindi con questa versione ottimizzata riusciamo a trovare in $O(n^2 \log^3 n)$ tempo il taglio minimo globale con probabilità $\geq 1 - 1/poly(n)$.

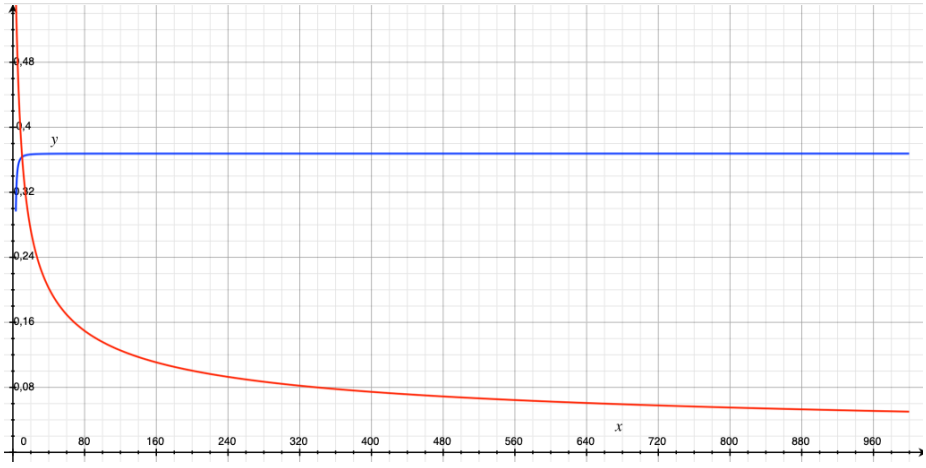

Figura 4.4: Probabilità di insuccesso al crescere di n

4.5.2 *Numero di tagli minimi*

L'analisi dell'algoritmo EdgeContraction permette di rispondere anche ad un'altra domanda: quanti tagli minimi ci sono?

In una rete di flusso è facile vedere che il numero di tagli sorgente-pozzo può essere esponenziale. Ad esempio consideriamo il grafo con nodi $V = \{s, t, v_1, v_2, \ldots, v_n\}$ e archi con capacità 1, (s, v_i) e (v_i, t). Tutti i tagli s-t devono avere grandezza n e un qualsiasi sottoinsieme S di $\{v_1, v_2, \ldots, v_n\}$ determina un taglio minimo $(\{s\} \cup S, \{t\} \cup V \setminus S)$. Si veda la Figura 4.5. Quindi esistono 2^n tagli minimi.

La situazione è diversa se prendiamo un grafo non direzionato. Se consideriamo un ciclo di n nodi, i tagli minimi hanno grandezza 2 e ce ne sono $\binom{n}{2}$ che si ottengono tagliando due archi qualsiasi, come mostrato nella Figura 4.6. È possibile trovare grafi con un numero di tagli minimi più grande? La risposta è no, e deriva dalla prova del Lemma 4.5.2.

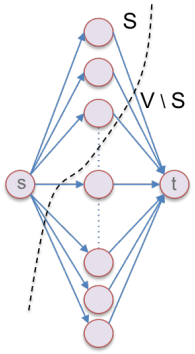

Figura 4.5: Taglio in un grafo s-t

Lemma 4.5.3 *In un grafo non direzionato con n nodi, ci sono al massimo* $\binom{n}{2}$ *tagli minimi globali*

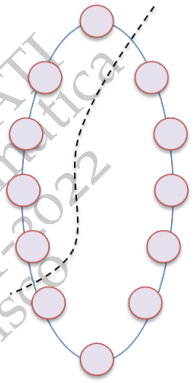

Figura 4.6: Taglio in un ciclo

Dimostrazione. Sia G il grafo e siano $C_1, \ldots, C_r$ tutti i tagli minimi globali. Sia α_i l'evento che C_i sia restituito dall'algoritmo EdgeContraction, e sia $\alpha = \bigcup_{i=1}^{r} \alpha_i$.

Nella prova del Lemma 4.5.2 abbiamo visto che $Pr[\alpha_i] \geq 1/\binom{n}{2}$. Poichè gli eventi α_i sono disgiunti, e poichè in ogni singola esecuzione dell'algoritmo viene prodotto un solo taglio, si ha che la probabilità dell'unione degli eventi è uguale alla somma delle probabilità dei singoli eventi e quindi

$$Pr[\alpha] = Pr[\bigcup_{i=1}^{r} \alpha_i] = \sum_{i=1}^{r} Pr[\alpha_i] \geq r/\binom{n}{2}.$$

Ovviamente si ha anche che $Pr[\alpha] \leq 1$ e quindi possiamo concludere che $r \leq \binom{n}{2}$. □

4.6 *Un algoritmo randomizzato per* MAX-3SAT.

Nel capitolo 1 abbiamo studiato il problema 3SAT ed abbiamo stabilito che è un problema $\mathcal{NP}$-completo. Ricordiamo il problema: abbiamo un insieme di k clausole $C_1, \ldots, C_k$, ognuna è la disgiunzione di 3 letterali deifiniti su n variabili $x_1, \ldots, x_n$. Il problema 3SAT è quello di stabilire se esiste un assegnamento di valori alle variabili in modo tale da rendere vere tutte le clausole. Un modo diverso di approcciare questo problema è quello di vederlo come un problema di ottimizzazione: trovare un assegnamento che renda vere il massimo numero possibile di clausole. Chiaramente anche questo problema è difficile, e di fatto è un problema NP-hard, in quanto è chiaro che 3SAT può essere ridotto MAX-3SAT. Il problema MAX-3SAT si presta ad essere l'oggetto di un algoritmo di approssimazione.

Anzi, questo è uno di quei casi dove l'algoritmo è di una semplicità estrema per cui non specifichiamo nemmeno lo pseudocodice: l'algoritmo assegna ad ogni singola variabile un valore a caso, 0 o 1 con probabilità ciascuno $1/2$. Quale è il numero atteso di clausole soddisfatte?

Lemma 4.6.1 *La soluzione fornita da un assegnamento casuale delle variabili ha un valore che dista dall'ottimo di un fattore moltiplicativo pari a* $7/8$.

DIMOSTRAZIONE. Sia Z una variabile casuale che denota il numero di clausole soddisfatte. Possiamo decomporre tale variabile nella somma di variabili Z_i dove ogni $Z_i = 1$ se la clausola C_i è soddisfatta, 0 altrimenti. Quindi si ha che $Z = Z_1 + Z_2 + \ldots + Z_k$. Il valore atteso di Z_i, $E[Z_i]$ è dato dalla probabilità che la clausola C_i sia vera. Questa probabilità è facile da calcolare. Infatti C_i è la disgiunzione di 3 letterali ed è vera se almeno uno dei 3 letterali ha il valore vero. Poichè tutte le variabili hanno valore vero con probabilità $1/2$ anche tutti i letterali, indipendentemente dal fatto che rappresentano una variabile o la sua negazione, hanno probabilità $1/2$ di essere veri e la stessa di essere falsi. L'unico caso in cui la clausola è falsa è quello in cui tutti i letterali sono falsi. Ciò avviene con probabilità $1/8$. Quindi la clausola C_i è vera con probabilità $7/8$, quindi $E[Z_i] = 7/8$. Usando la linearità del valore atteso si ha che $E[Z] = E[Z_1] + E[Z_1] + \ldots + E[Z_k] = \frac{7}{8}k$.

D'altra parte l'assegnamento ottimale può soddisfare al massimo k clausole semplicemente perché non ce ne sono di più. □

La prova che abbiamo appena fornito in realtà ci dice anche di più di quello che abbiamo asserito nell'enunciato del lemma. Infatti poichè il valore atteso del numero di clausole soddisfatte per un assegnamento casuale è $7/8$ di quello ottimale, è necessario che esista almeno un assegnamento per il quale effettivamente almeno $7/8$ delle clausole sono soddisfatte (altrimenti il valore medio dovrebbe essere più basso). Pertanto si ha:

Lemma 4.6.2 *Per una qualsiasi istanza di* 3SAT *esiste un assegnamento delle variabili che rende vere almeno* $7/8$ *delle clausole.*

L'asserzione del precedente lemma è abbastanza sorprendente in quanto non ha nulla a che vedere con la randomizzazione. Ma l'abbiamo ottenuta utilizzando l'analisi di un

algoritmo randomizzato. Questo tipo di risultato è abbastanza comune nell'area della combinatorica: si prova l'esistenza di oggetti con determinate caratteristiche mostrando che una costruzione casuale li produce con una probabilità non nulla. In questi casi si parla di *metodo probabilistico*.

Un'altra osservazione riguardo la precedente asserzione: ogni istanza di 3SAT con al massimo 7 clausole è soddisfattibile. Infatti per il lemma precedente esiste un assegnamento in cui almeno 7/8 delle clausole sono vere. Ma se l'istanza ha $k \leq 7$ clausole, si ha che $\frac{7}{8}k > k - 1$ e quindi per quel particolare assegnamento tutte le clausole sono vere.

Sappiamo dunque che un assegnamento casuale produce un assegnamento che soddisfa mediamente un numero di clausole pari a $\frac{7}{8}k$. Tuttavia non è per nulla garantito che assegnando casualmente i valori alle variabili vengano soddisfatte tante clausole; $\frac{7}{8}k$ è un valore medio. Se un assegnamento non soddisfa almeno $\frac{7}{8}k$ clausole possiamo ripetere l'assegnamento fino a quando non ne troviamo uno (sappiamo che esiste) che soddisfi tante clausole. Ma quante volte dobbiamo ripetere l'assegnamento per avere almeno $\frac{7}{8}k$ clausole vere?

Lemma 4.6.3 *Per trovare un assegnamento che soddisfi almeno $\frac{7}{8}k$ clausole, il numero medio di ripetizioni che dobbiamo utilizzare è $8k$.*

DIMOSTRAZIONE. Consideriamo come esperimento l'assegnazione dei valori alle variabili e come successo il fatto che ci siano almeno $\frac{7}{8}k$ clausole vere.

Indichiamo con p_j la probabilità che esattamente j clausole siano vere, per $j = 0, 1, \ldots, k$ si ha che il numero medio di clausole vere è

$$0 \cdot p_0 + 1 \cdot p_1 + \ldots k \cdot p_k = \sum_{j=0}^{k} j p_j.$$

Sappiamo già che tale numero è $\frac{7}{8}k$, pertanto si ha che

$$\sum_{j=0}^{k} j p_j = \frac{7}{8}k.$$

D'altra parte, la probabilità di successo dell'esperimento è data da

$$p = \sum_{j \geq \frac{7}{8}k} p_j.$$

Definiamo k' come l'intero più grande tale che $k' < \frac{7}{8}k$. Prima di proseguire analizziamo la relazione fra k e k'. Nella seguenmte tabella sono riportati i valori per alcuni k piccoli.

k	$\frac{7}{8}k$	k'	$\frac{7}{8}k - k'$
0	0	-1	8/8
1	7/7	0	7/8
2	14/8	$1 = 8/8$	6/8
3	21/8	$2 = 16/8$	5/8
4	28/8	$3 = 24/8$	4/8
5	35/8	$4 = 32/8$	3/8
6	42/8	$5 = 40/8$	2/8
7	49/8	$6 = 48/8$	1/8
8	56/8	$6 = 48/8$	8/8
9	63/8	$7 = 56/8$	7/8
...	...	...	...
14	98/8	$12 = 96/8$	2/8
15	105/8	$13 = 104/8$	1/8
16	112/8	$13 = 104/8$	8/8
...	...	...	...

Non è difficile dimostrare che $\frac{1}{8} \leq \frac{7}{8}k - k'$. Un semplice studio della funzione porta a dimostrare che la funzione ha l'andamento mostrato nella Figura 4.7.

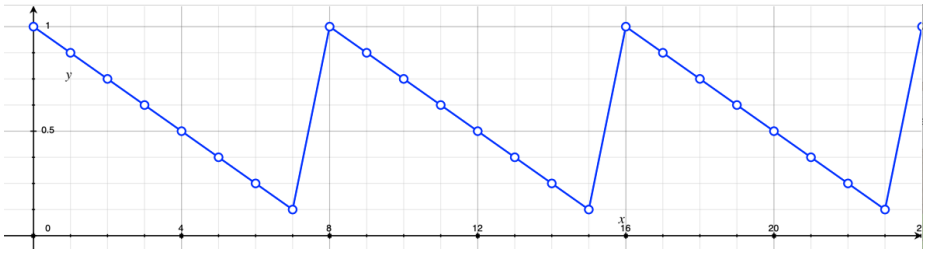

Figura 4.7: Funzione $\frac{7}{8}k - k'$

Si ha che:

$$\begin{aligned}
\frac{7}{8}k &= \sum_{j=0}^{k} jp_j \\
&= \sum_{j<7k/8} jp_j + \sum_{j\geq 7k/8} jp_j \\
&\leq \sum_{j<7k/8} k'p_j + \sum_{j\geq 7k/8} kp_j \\
&= k' \sum_{j<7k/8} p_j + k \sum_{j\geq 7k/8} p_j \\
&= k'(1-p) + kp \\
&\leq k' + kp
\end{aligned}$$

Quindi

$$\frac{7}{8}k - k' \leq kp.$$

Ricordando che $\frac{1}{8} \leq \frac{7}{8}k - k'$, pertanto

$$\frac{1}{8} \leq kp$$

e quindi

$$p \geq \frac{1}{8k}.$$

Dal Lemma 4.3.1 si ha che il numero di ripetizioni prima di avere successo è $1/p$, quindi è $\leq 8k$. □

4.7 *Note bibliografiche*

Fonti consultate: KT2014 [20], A2016 [2]. L'algoritmo EdgeContraction è dovuto a Karger [18] mentre la versione ottimizzata a Karger e Stein [19].

4.8 *Esercizi*

1. Nell'esempio del gioco delle 3 carte abbiamo usato un algoritmo randomizzato che gira sempre tutte e tre le carte. Il gioco però ammette anche che si giri una sola carta o due carte, il che permetterebbe di risparmiare sulla cifra necessaria per giocare. Ricordando che si paga 1€per ogni giocata e che la vincita vale 2€, perchè abbiamo usato un algoritmo che gira tutte e tre le carte?

2. Consideriamo il gioco delle 4 carte che è come quello delle 3 carte, ma con una carta in più. Anche in questo caso si paga 1€ per girare una singola carta e si vincono 2€ quando si scopre la carta vincente. Supponendo di dover decidere di fissare un numero di carte da scoprire in ogni partita, quale è la strategia migliore e perchè?

3. Consideriamo un ulteriore variante: questa volta giochiamo con 4 carte, ed una vincita di 3€, mentre il costo per girare una carta è sempre di 1€. Cosa cambia? Quale è la strategia migliore?

4. Consideriamo (per l'ultima volta!!!!) il gioco delle 3 carte e questa volta generalizziamo ad n carte e giochiamo m partite. Il costo per scoprire una singola carta è di 1€ e la vincita per l'individuazione della carta vincente è di v€, con v che può assumere i valori da 2 a n. Ora le strategie possibili sono n: girare k carte, dove k può essere un valore fra 1 e n. Quale è la strategia migliore?

5. Fornire un esempio di input per il quale QuickSort genera un'alternanza fra caso sfortunato (partizione completamente sbilanciata) e caso frotunato (partizione completamente bilanciata) nella sequenza di chiamate ricorsive. Più precisamente nel primo livello dell'albero della ricorsione si è fortunati, nel secondo livello si è sfortunati (su tutte le chiamate ricorsive), nel terzo livello si è fortunati (su tutte le chiamate ricorsive), e così via. Motivare la risposta.

6. Fornire un algoritmo RandomSelectBil, simile a RandomSelect, sfruttando la stessa idea con la quale si è progettato RandomQuickSortBil partendo da RandomQuickSort. Analizzare il tempo di esecuzione.

7. Un'azienda con n dipendenti deve pianificare l'esecuzione di un insieme di m lavori $w_1, \ldots, w_m$, ognuno dei quali deve essere svolto da due dipendenti. L'assegnazione

dei lavori ai dipendenti è fissata, cioè per ogni lavoro w_i, sono stati assegnati due dipendenti $d_1(w_i)$ e $d_2(w_i)$. Non c'è un vincolo sul numero di lavori a cui un dipendente può essere assegnato. Per svolgere i lavori sono di aiuto 3 specifiche competenze: se il lavoro viene svolto da due dipendenti con competenze diverse il tempo necessario a svolgerlo è minore. Poichè l'azienda deve investire denaro per far acquisire ad ognuno dei dipendenti una delle 3 competenze, deve decidere quale competenza far acquisire ad ognuno dei dipendenti. Ovviamente l'obiettivo è quello di massimizzare il numero di lavori che verranno svolti da due dipendenti con competenze diverse.

Si fornisca un algoritmo randomizzato che produce una soluzione $(2/3)$-approssimata. Fornire l'analisi.

8. Consideriamo il problema MAXSAT che generalizza il problema MAX-3SAT studiato in questo capitolo: massimizzare il numero di clausole vere in una formula ϕ; le clausole della formula possono avere un numero qualsiasi di letterali. Per MAXSAT abbiamo visto un algoritmo randomizzato che forniva un'approssimazione di $7/8$. Analizzare il comportamento dell'algoritmo per MAXSAT.

9. Consideriamo il problema dell'individuazione di un taglio minimo che divide una sorgente s da un destinazione t. Potremmo utilizzare l'algoritmo EdgeContraction se si fa in modo che s e t non vengano mai contratti. Basterà, dopo ogni contrazione, rimuovere tutti gli archi che connettono s e t. Analizzare questo algoritmo e dire se può essere utile usandolo ripetutamente come EdgeContraction.