Capitolo 3
Complessità computazionale

Si è già accennato nel primo capitolo come nella teoria dell'ottimizzazione l'interesse ricada sul progetto di algoritmi per la risoluzione di problemi di ottimizzazione piuttosto che non sulla 'soluzione' medesima. Ovviamente in fase finale di applicazione ciò che conta è la soluzione. La teoria dell'ottimizzazione si occupa invece della fase preliminare, cioè di come generare la soluzione, ed è naturale pensare di adottare dei metodi di risoluzione che possano essere ripetutamente applicati anche se i dati del problema variano.

Non è certamente pensabile un metodo universale che sia in grado di risolvere qualsiasi problema di ottimizzazione. Tale metodo, peraltro di improbabile esistenza, non potrebbe sfruttare le peculiarità strutturali dei singoli problemi e quindi 'funzionerebbe' male. È quindi naturale identificare classi di problemi molto simili tra loro e costruire dei metodi di risoluzione in grado di affrontare nel modo più efficiente possibile tutti i problemi della classe.

Siccome la risoluzione avviene tramite calcolatore, i metodi di risoluzione devono essere di natura algoritmica, cioè devono operare entro un insieme strutturato di regole formalmente definite. Ogni algoritmo, per portare a termine il calcolo richiesto, richiede risorse di tempo e di spazio il cui ammontare prende il nome di *complessità computazionale dell'algoritmo*.

La teoria della complessità computazionale si occupa della valutazione della complessità di un algoritmo, ma non si limita a questo. Si vuole sapere inoltre se una certa complessità sia dovuta non tanto alle peculiarità di un certo algoritmo, quanto a delle caratteristiche intrinseche alla classe di problemi per cui l'algoritmo è progettato. Si parla allora di complessità computazionale di un problema. È questo forse, dal punto di vista dell'ottimizzazione, l'aspetto più interessante e utile della teoria perché permette, per certi problemi, di fare affermazioni a priori sulla complessità di un possibile algoritmo risolutivo e pertanto ne guida efficacemente il progetto.

Per poter valutare la complessità computazionale (di un algoritmo o di un problema) è però necessario definire in modo formale cosa si intenda per 'algoritmo', per 'risorsa' (sia temporale che spaziale) e per 'problema'. In questa sede ci si limiterà a fornire soltanto le nozioni di base della teoria della complessità computazionale, che sono comunque sufficienti a determinare la complessità della maggior parte dei problemi di ottimizzazione. Il lettore può trovare materiale analogo in Papadimitriou e Steiglitz [1983] oppure in Maffioli [1990]. Per un maggiore approfondimento è consigliabile il testo tuttora valido di Garey e Johnson [1979]. Infine per una trattazione completa sull'argomento si consigliano il libro di testo di Papadimitriou [1995] e il saggio di Johnson [1990]. È anche raccomandabile la monografia di Blum e al. [1998] che estende la complessità computazionale ai numeri reali (ma si veda al riguardo anche la breve monografia di Lovász [1986]).

3.1. Algoritmi

Il modello formale di algoritmo che si usa nella teoria della complessità computazionale è la macchina di Turing. Il matematico inglese Alan Turing definì nel 1936, molto prima dell'avvento dei calcolatori nel 1947, un modello astratto di calcolo che da lui prese appunto il nome di macchina di Turing (Turing [1936]). Il fatto sorprendente è che un tale modello è perfettamente adeguato, almeno dal punto di vista teorico, ad eseguire senza perdita di efficienza ogni calcolo eseguibile su macchine reali.

Una trattazione approfondita della teoria richiederebbe di presentare tutti i risultati riferiti alla macchina di Turing. Tuttavia, proprio per l'equivalenza teorica esistente fra una macchina di Turing e una normale macchina ad accesso casuale, i risultati principali della teoria possono essere presentati facendo riferimento ai più familiari concetti di macchina ad accesso casuale e di algoritmi esprimibili in linguaggi ad alto livello, quali ad esempio il Pascal oppure il C.

Gli algoritmi operano su simboli che possono essere tratti da un qualsiasi alfabeto finito Σ, sia questo l'alfabeto binario $\{0, 1\}$ oppure l'insieme dei simboli presenti su una tastiera di calcolatore. Con i simboli di Σ si possono formare stringhe di lunghezza arbitraria. L'insieme delle stringhe di simboli di Σ di lunghezza finita (ma arbitraria) viene indicato con Σ^*.

Un algoritmo è una particolare funzione $\Sigma^* \to \Sigma^*$ che trasforma stringhe, dette d'ingresso, in stringhe, dette d'uscita. La trasformazione deve avvenire secondo uno schema prefissato di operazioni tratte da un insieme finito e codificato, come avviene ad esempio con un programma scritto in un linguaggio ad alto livello. Nel modello sequenziale di algoritmo le operazioni vengono eseguite una alla volta. Per essere tale, un algoritmo deve terminare, cioè richiedere un numero finito di operazioni, per ogni possibile stringa d'ingresso.

Il 'tempo' di esecuzione dell'algoritmo viene assimilato al numero di operazioni eseguite dall'inizio della lettura della stringa d'ingresso alla fine della scrittura della stringa d'uscita. Questo è equivalente ad assumere tempo unitario (secondo un'unità di misura che non serve specificare) per ogni operazione. Si indichi con $\tau(s)$ il tempo di esecuzione dell'algoritmo con stringa d'ingresso s.

Lo 'spazio' usato in esecuzione dall'algoritmo è la massima quantità di memoria interna richiesta per immagazzinare risultati parziali necessari alle operazioni successive, espressi sotto forma di stringhe di Σ^*. Si tenga presente che, qualora i risultati parziali non servano più, possono essere cancellati e la memoria resa nuovamente disponibile. Per convenzione non si tiene conto dello spazio necessario alle stringhe d'ingresso e d'uscita (memoria esterna). Si indichi con $\sigma(s)$ lo spazio usato in esecuzione dall'algoritmo con stringa d'ingresso s.

Va osservato che, siccome ogni operazione di scrittura sulla memoria interna è di fatto un'operazione dell'algoritmo, si ha $\tau(s) \geq \sigma(s)$. Sembrerebbe inoltre che $\tau(s)$ possa essere arbitrariamente più grande di $\sigma(s)$. Ciò non è però possibile. Infatti vi possono essere al massimo $K := |\Sigma|^{\sigma(s)}$ configurazioni diverse di memoria, e, supponendo che l'algoritmo sia scritto con H istruzioni e che la stringa d'ingresso sia già stata letta, se l'algoritmo richiedesse più di $K\,H$ operazioni, necessariamente ce ne sarebbero due corrispondenti alla medesima istruzione con la medesima configurazione di memoria, e le operazioni fra le due istruzioni si ripeterebbero in ciclo, impedendo all'algoritmo di terminare. Quindi un limite sulla memoria implica un limite, anche se esponenzialmente elevato, al tempo. Importante è notare che una limitazione logaritmica sulla memoria implica una limitazione polinomiale sul tempo (ma non viceversa ovviamente).

3.2. Complessità di un algoritmo

È naturale che per elaborare una quantità maggiore di dati un algoritmo richieda maggiori risorse. La nozione di 'efficienza' di un algoritmo viene appunto misurata facendo riferimento al modo in cui crescono $\tau(s)$ e $\sigma(s)$ al crescere di s. Prima di definire una tale misura è utile richiamare le seguenti definizioni:

3.1 Definizione. *Sia $f : N \to N$. Si definiscono i seguenti insiemi:*

$$O(f(n)) := \left\{ g \in N^N : \textit{esistono } \hat{k},\, c > 0 \textit{ tali che } g(k) \leq c\, f(k),\ \textit{per } k \geq \hat{k} \right\}$$
$$\Omega(f(n)) := \left\{ g \in N^N : f \in O(g(n)) \right\}$$
$$\Theta(f(n)) := O(f(n)) \cap \Omega(f(n))$$

Se $g \in O(f)$ si dice che 'g è o di *f'. Se $g \in \Omega(f)$ si dice che 'g* è omega di *f'. Se $g \in \Theta(f)$ si dice che 'g* è theta di *f'.* ∎

Quindi $O(f)$ è l'insieme delle funzioni che asintoticamente non crescono più velocemente di f, e dire che $g \in O(f)$ corrisponde a stabilire una limitazione superiore alla crescita di g. Invece $\Omega(f)$ è l'insieme delle funzioni che asintoticamente non crescono più lentamente di f e dire che $g \in \Omega(f)$ corrisponde a stabilire una limitazione inferiore alla crescita di g. Funzioni in $\Theta(f)$ sono equivalenti ad f e fra loro. La maggior parte delle valutazioni che si fanno nella teoria della complessità computazionale riguarda limitazioni superiori, non solo perché sono utili, ma anche perché sono più facili da ottenere. Anche la conoscenza delle limitazioni inferiori è importante, però queste si riescono ad ottenere molto raramente.

Date le funzioni $\tau(s)$ e $\sigma(s)$, si definiscono funzioni analoghe in dipendenza però dalla lunghezza $|s|$ della stringa s:

$$\hat{\tau}(n) := \max_{s:|s|\leq n} \tau(s) \qquad\qquad \hat{\sigma}(n) := \max_{s:|s|\leq n} \sigma(s) \tag{3.1}$$

Si noti che questa definizione tiene conto del peggior comportamento dell'algoritmo limitatamente a stringhe di lunghezza non superiore ad un fissato valore. Questo tipo di misura sull'algoritmo fornisce una garanzia di qualità, perché per definizione non potrà mai avvenire che l'algoritmo richieda tempo superiore a $\hat{\tau}(n)$ oppure spazio superiore a $\hat{\sigma}(n)$. Si può ragionevolmente obiettare che, se un algoritmo deve essere ripetuto molte volte, è forse più interessante conoscere la quantità media di risorse richiesta anziché la massima, o forse addirittura una quantità media per una opportuna distribuzione di probabilità delle stringhe d'ingresso. Due sono essenzialmente le ragioni per cui si usa (3.1): limitazioni di caso peggiore sono di gran lunga più facili da ottenere ed è molto difficile giustificare teoricamente l'uso di una particolare distribuzione di probabilità sulle stringhe d'ingresso.

Una valutazione diretta di $\hat{\tau}(n)$ o di $\hat{\sigma}(n)$ è difficile in generale. Quello che si riesce a fare è trovare delle funzioni f, di solito elementari quali polinomi e logaritmi, tali che si possa affermare $\hat{\tau} \in O(f)$ oppure $\hat{\sigma} \in O(f)$.

3.2 Definizione. *Se esiste un polinomio p tale che $\hat{\tau} \in O(p)$, un algoritmo si dice* polinomiale. *Se invece per ogni polinomio p si ha $\hat{\tau} \notin O(p)$, un algoritmo si dice* non polinomiale. ∎

3.3 Definizione. *Se esiste un polinomio p tale che $\hat{\sigma} \in O(p)$, un algoritmo si dice* polinomiale nello spazio. *Se invece per ogni polinomio p si ha $\hat{\sigma} \notin O(p)$, un algoritmo si dice* non polinomiale nello spazio. ∎

Data la maggior importanza della risorsa tempo rispetto alla risorsa spazio nella valutazione della complessità computazionale, si è scelto di definire semplicemente 'polinomiale' un algoritmo polinomiale nel tempo. Occasionalmente per maggior enfasi si userà la dizione piena 'polinomiale nel tempo'. Gli algoritmi polinomiali (e analogamente gli algoritmi polinomiali nello spazio) si possono classificare ulteriormente nelle seguenti classi di complessità:

— *costanti* se esiste una costante C tale che $\hat{\tau}(n) \leq C$ per ogni n;
— *logaritmici* se $\hat{\tau}(n) \in O(\log n)$;
— *polilogaritmici* se esiste una costante C tale che $\hat{\tau}(n) \in O(\log^C n)$;
— *lineari* se $\hat{\tau}(n) \in O(n)$;
— *quadratici* se $\hat{\tau}(n) \in O(n^2)$.

Si noti che, in base alle definizioni, un algoritmo costante è anche logaritmico, polilogaritmico, ecc. e che ogni algoritmo appartenente ad una delle categorie dell'elenco appartiene anche a quelle successive. Trattandosi di una limitazione superiore ha senso usare quella più stretta per cui, ad esempio, non si dirà mai di un algoritmo lineare che è quadratico. La classificazione data è quella a cui si fa normalmente riferimento. Tuttavia si può dimostrare che si possono operare classificazioni arbitrariamente fini. Inoltre gli algoritmi (polinomiali e non) vengono detti:

— *subesponenziali* se esiste una costante C tale che $\hat{\tau}(n) \in O(n^{\log^C n})$;
— *esponenziali* se esiste una costante C tale che $\hat{\tau}(n) \in O(2^{n^C})$;
— *illimitati* se esiste n tale che $\hat{\tau}(n) = \infty$.

3.4 Esercizio. Dimostrare che le classi elencate sono incluse strettamente una nell'altra. ∎

3.5 Esercizio. Fra quali classi di complessità si inseriscono le classi $O(n/\log n)$, $O(n \log n)$, $O(2^n/n)$? ∎

La complessità di un algoritmo è la più piccola classe di complessità a cui appartiene, con riferimento alla risorsa tempo. Se si fa riferimento alla risorsa spazio si parla di complessità nello spazio.

La distinzione operata nella definizione 3.2 fra algoritmi polinomiali e non polinomiali formalizza il concetto di algoritmo efficiente. In altri termini una limitazione polinomiale (nel tempo, non nello spazio) è sinonimo di efficienza. Questa definizione può sollevare qualche obiezione. Ad esempio, se $n = 1000$, un algoritmo di complessità $O(n^5)$ è probabilmente inutilizzabile mentre un altro di complessità $O(2^{0.01\,n})$ non crea problemi (come si vedrà, proprio nella programmazione lineare, tema centrale dell'ottimizzazione, si verifica la sorprendente eccezione che un algoritmo non polinomiale è più 'efficiente' di uno polinomiale, ma è forse l'unico caso anche se significativo). Inoltre può sembrare inappropriato mettere in un'unica classe di efficienza algoritmi logaritmici o lineari assieme ad algoritmi polinomiali senza limitazione di grado. In pratica c'è una differenza abissale fra $O(\log n)$ e $O(n^5)$.

Tuttavia vi sono valide ragioni per giustificare la definizione data di efficienza. Una prima ragione è teorica. La classe dei polinomi è chiusa rispetto alle operazioni di somma, moltiplicazione e composizione e quindi, come vedremo, sono permesse affermazioni di portata molto generale, probabilmente non possibili per altre classi, che danno luogo ad una teoria utile ed elegante. Dal punto di vista pratico bisogna dire che algoritmi polinomiali con gradi molto elevati (ad es. 20) non sono stati probabilmente mai proposti. In quei casi in cui inizialmente si era trovato un algoritmo polinomiale di grado elevato, ricerche successive avevano immediatamente abbattuto la complessità a valori con grado molto più basso. Di fatto la maggior parte degli algoritmi polinomiali di uso corrente ha un grado non superiore a tre. Un'altra considerazione di tipo pratico riguarda l'impatto sulle prestazioni di un algoritmo dovuto ad un miglioramento tecnologico che acceleri di un fattore ad esempio 1000

l'esecuzione di un'operazione. Nel medesimo intervallo di tempo si risolvono problemi 1000 volte più grandi con un algoritmo lineare, 32 volte più grandi con uno quadratico e 10 volte più grandi con uno cubico. Però se l'algoritmo ha una complessità $\Theta(2^n)$ si passa da n a $n+10$, con un semplice fattore additivo anziché moltiplicativo. In altre parole la tecnologia non sarà mai d'aiuto per rendere praticabile un algoritmo non polinomiale.

3.3. Numeri e operazioni sui numeri

Nel calcolo scientifico è prassi costante operare su numeri, siano essi indifferentemente interi, razionali o reali. Il fatto che un numero irrazionale non sia in generale rappresentabile con una stringa finita di simboli, a differenza dei numeri interi o razionali, non costituisce normalmente un problema fondamentale, in quanto si può operare con approssimazioni che risultano soddisfacenti se vengono prese opportune precauzioni. In questo tipo di calcoli, in cui non è mai richiesto di operare con un numero elevato di cifre significative (come avviene invece in crittografia ad esempio), l'onere computazionale indotto dalle operazioni aritmetiche su un numero è praticamente costante, indipendentemente dal valore del numero. In quest'ottica ha senso allora supporre che un numero contribuisca alla lunghezza della stringa d'ingresso con una quantità fissa. Questa codifica idealizzata di un numero prende il nome di *codifica aritmetica* e ad essa fa riscontro la pratica di assegnare ad ogni numero, indipendentemente dal suo valore, una quantità fissa nella memoria di un calcolatore. Scelta la codifica aritmetica, le operazioni aritmetiche sui numeri sono necessariamente considerate operazioni elementari che richiedono sempre un'unità di tempo (*misura uniforme*). Ad esempio, secondo la codifica aritmetica e la misura uniforme, eseguire il prodotto scalare di due vettori di n elementi richiede n moltiplicazioni e $n-1$ somme ed ha in ogni caso complessità temporale $O(n)$.

Se invece si vuole valutare la complessità computazionale di un algoritmo in modo rigoroso (come se si trattasse di una macchina di Turing), bisogna considerare più in dettaglio come un numero viene codificato e come i simboli della codifica vengono manipolati. Innanzitutto si devono escludere i numeri reali dall'ambito dei dati. Solo numeri razionali ed interi sono ammessi. Usando una rappresentazione in base β, un numero intero a ha bisogno di $\lceil \log_\beta |a| + 1 \rceil + 1$ simboli (tenendo conto anche del segno). Questa codifica prende il nome di *codifica normale* (o *binaria* se $\beta = 2$).

A questo livello di dettaglio possiamo considerare come elementari solo le operazioni sui simboli rappresentanti un numero. Quindi una somma di due interi a e b (supponiamo per semplicità di notazione che le loro stringhe richiedano lo stesso numero L di simboli) non ha complessità costante ma lineare, in quanto può richiedere fino a $2\,L$ operazioni (tenendo conto anche dei possibili riporti). Il prodotto degli stessi interi non ha complessità costante ma quadratica $O(L^2)$. Questa misura della complessità delle operazioni aritmetiche prende il nome di *misura logaritmica*.

Secondo la codifica normale e la misura logaritmica, il prodotto scalare di due vettori di n elementi, tutti esprimibili con lo stesso numero L di simboli, richiede un numero di operazioni $O(n\,L + n\,L^2)$. Siccome la lunghezza della stringa rappresentante i vettori è $S = \Omega(n\,L)$, la complessità riferita alla lunghezza della stringa d'ingresso è $O(S(1 + S/n))$. Si tratta di una complessità quadratica per ogni valore fissato di n e lineare per ogni valore fissato di L. Il caso peggiore asintotico porta allora ad una complessità quadratica.

A questo punto si può notare che considerare la misura logaritmica anziché quella uniforme non altera la natura polinomiale di un algoritmo. Inoltre il senso pratico della valutazione asintotica riguarda di solito quanti sono i numeri coinvolti nel calcolo e non quanto sono

grandi. Per questi motivi risulta più 'semplice' adottare la misura uniforme e la codifica aritmetica nella maggior parte dei casi.

Vi sono tuttavia algoritmi in cui il numero di operazioni, anche secondo la misura uniforme, dipende dalla grandezza dei numeri. Se ad esempio si deve operare una ricerca binaria sugli interi fra 1 e K, sono necessarie $\log_2 K$ operazioni di confronto e valutazione e quindi la complessità computazionale non può non dipendere dalla grandezza dei numeri. In questo caso adottare la codifica normale (pur adottando la misura uniforme) è obbligatorio, perché con la codifica aritmetica si avrebbe che il numero di operazioni è arbitrariamente elevato a fronte di una codifica costante dei dati d'ingresso, portando quindi ad una complessità illimitata.

Il fatto che i dati in ingresso occupino meno spazio con la codifica aritmetica che con quella normale rende apparentemente più 'lento' un algoritmo, se valutato con la codifica aritmetica. Sembra quindi che un algoritmo, polinomiale con la codifica aritmetica, lo debba essere anche rispetto a quella normale. Tuttavia lo stesso ragionamento applicato ai dati d'uscita anziché d'ingresso, come anche ai dati intermedi, produce il risultato opposto, come si vede dal semplice algoritmo $a_k := a_{k-1}^2$, $k = 1, \ldots, n$, con dati d'ingresso a_0 e n, e dato d'uscita a_n. L'algoritmo ha complessità $O(n)$ con la codifica aritmetica, ma $\Omega(2^n \log a_0)$, e quindi esponenziale, con la codifica normale e la misura logaritmica.

Siccome si vuole attribuire importanza al comportamento di un algoritmo rispetto alla codifica aritmetica, nell'idea che la codifica aritmetica penalizza un algoritmo più di quella normale, dobbiamo allora escludere la possibilità di scritture troppo lunghe sia per i dati in uscita che per tutti i dati intermedi. Si ha quindi la seguente:

3.6 Definizione. *Un algoritmo che richieda tempo polinomiale rispetto alla codifica aritmetica e spazio polinomiale rispetto alla codifica normale si dice* fortemente polinomiale. ∎

Si noti che un algoritmo fortemente polinomiale è anche polinomiale in virtù dell'ipotesi sulla complessità spaziale.

3.7 Esempio. Si consideri il calcolo del massimo comun divisore (MCD) di due interi a e b (supponiamo $a > b$) eseguito con l'algoritmo di Euclide. Come è noto, si tratta di calcolare il resto della divisione di a con b. Se il resto è 0, il MCD è b, altrimenti si calcola il MCD di b e del resto (che è necessariamente minore di b) e si procede ricorsivamente fino a che il resto è nullo. A questo punto l'ultimo resto positivo è il MCD di a e b. Posto $c_0 := a$ e $c_1 := b$ l'algoritmo produce una successione di numeri $c_2, \ldots, c_{n+1}$, e $q_1, \ldots, q_n$ tali che

$$c_{k-1} = c_k\, q_k + c_{k+1},\ k = 1, \ldots, n \qquad\qquad c_{n+1} = 0$$

e c_n è il MCD di a e b. Il lettore può per esercizio verificare la correttezza dell'algoritmo. Vogliamo ora valutarne la complessità e a questo scopo si deve capire come n (numero di divisioni da effettuare) dipenda da a e b. Ridenominiamo la successione nel seguente modo: $d_0 := c_{n+1} = 0$, $d_1 := c_n$, $d_2 := c_{n-1}$, $d_n := c_1 = b$, $d_{n+1} := c_0 = a$. Notiamo che $q_k \geq 1$ siccome $c_{k+1} < c_k$, e quindi :

$$d_{k+1} \geq d_k + d_{k-1},\ k = 1, \ldots, n \tag{3.2}$$

Siano f_k i numeri di Fibonacci generati dalla ricorsione $f_0 := 0$, $f_1 := 1$, $f_{k+1} := f_k + f_{k-1}$, $k \geq 1$, e si noti che, da $d_0 = f_0$, $d_1 \geq f_1$, applicando ricorsivamente (3.2) si ha $d_k \geq f_k$, $\forall k$. Si sa che

$$f_k = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^k - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^k$$

Siccome il secondo dei due termini è minore di 1 e tende a 0 si ha $f_n = \Theta(\alpha^n)$ con $\alpha > 1$. Da $b = d_n > f_n$ si ha $n \in O(\log b)$. Adottando la codifica normale e la misura uniforme, la complessità dell'algoritmo di Euclide è quindi lineare, visto che ogni divisione costa soltanto un'unità temporale. Se invece usiamo la misura logaritmica, la complessità deve tener conto anche delle operazioni necessarie per calcolare le divisioni ($O(L^2)$ per ogni divisione con numeri che richiedono L simboli).

Se però usiamo la codifica aritmetica, i dati d'ingresso sono costanti (due numeri) mentre il numero di divisioni cresce senza limiti. L'algoritmo di Euclide quindi non è fortemente polinomiale. ∎

Vi è una terza possibilità, anche se solo teorica, di codificare un numero. In un modo molto primitivo un intero potrebbe essere codificato con un numero di simboli uguali pari al suo valore. Questa codifica prende il nome di codifica *unaria*. In questo caso la lunghezza di una stringa d'ingresso viene artificialmente ingrandita e l'algoritmo risulta apparentemente più veloce. L'introduzione della codifica unaria ha l'unico scopo di capire se alcuni algoritmi non polinomiali possano diventare polinomiali con la nuova codifica.

3.8 Definizione. *Un algoritmo che richieda tempo polinomiale con la codifica unaria, ma non con la codifica normale, si dice* pseudopolinomiale. ∎

3.9 Esempio. Un elementare algoritmo per provare che un numero a è primo consiste nel provare a dividerlo per $2, 3, \ldots, \lfloor\sqrt{a}\rfloor$ e verificare che non si ottiene mai resto nullo. La complessità è quindi $O(\sqrt{a})$, che *non* è polinomiale nonostante le apparenze. Il numero a richiede L simboli in base β con $L \in \Theta(\log_\beta a)$, e allora la complessità che si ottiene, $O(\beta^{L/2})$, non è polinomiale. L'algoritmo è dunque pseudopolinomiale. ∎

3.4. Problemi e istanze

Finora il termine 'problema' è stato usato in modo informale senza badare all'ambiguità di significato che deriva dall'intendere a volte un singolo problema e a volte una intera classe di problemi simili tra loro. Nella teoria della complessità computazionale viene riservato il termine istanza per indicare un singolo 'problema' con tutti i dati fissati, mentre il termine problema è riservato per indicare una classe di istanze simili tra loro. Quindi, quando si parla del problema della programmazione lineare, si intende l'insieme dei 'problemi' in cui la funzione obiettivo è lineare e i vincoli sono espressi da equazioni e/o diseguaglianze lineari, mentre un'istanza di un problema di programmazione lineare è l'insieme dei dati che caratterizzano una specifica funzione obiettivo e degli specifici vincoli. Un algoritmo è allora progettato per risolvere tutte le istanze di un determinato problema.

La definizione appena data di problema è ancora informale. Per una definizione formalmente corretta bisogna fare riferimento alle stringhe di ingresso e di uscita che rappresentano, la prima i dati di un'istanza e la seconda la soluzione (l'ottimo per problemi di ottimizzazione) dell'istanza.

3.10 Definizione. *Un* problema *è un insieme X di coppie ordinate $(I, U) \in \Sigma^* \times \Sigma^*$, con la proprietà che la proiezione di X sulla prima componente è Σ^*. La stringa d'ingresso I viene detta* istanza *e la stringa d'uscita U viene detta* risposta *dell'istanza.* ∎

Questa definizione astratta di problema richiede alcuni commenti. Come prima cosa si noti che ogni istanza, non importa come formulata inizialmente (matrici, grafi ecc.) viene tradotta in una rappresentazione simbolica, che è la stringa d'ingresso, ed è a questa che si fa

riferimento per la valutazione della complessità. La proprietà espressa nella definizione, che ogni stringa debba figurare come ingresso di qualche istanza, può forse sembrare fuori luogo. Infatti, fissata una regola di codifica dei dati di un problema (ad esempio la descrizione di un grafo), vi sono alcune stringhe che effettivamente corrispondono alla codifica di un'istanza. Altre stringhe, probabilmente la maggioranza, sono stringhe senza alcun significato per il problema in esame non rappresentando nessuna codifica. Conviene comunque avere una definizione estesa di problema, ammettendo in ingresso anche stringhe senza significato. In tal caso si può sempre supporre che la corrispondente stringa di uscita corrisponda ad un messaggio di 'ingresso senza senso'. Negli altri casi, se il problema è di ottimizzazione, la risposta dell'istanza può contenere l'ottimo, oppure un messaggio di problema non ammissibile oppure illimitato. Si noti ancora che in X possono essere presenti coppie (i, u) e (i, u') con $u \neq u'$ (ad esempio, se vi sono più ottimi per la stessa istanza i, u e u' potrebbero essere due ottimi diversi).

Un problema, come definito in 3.10, prende il nome anche di *problema di ricerca*, in quanto si tratta di 'trovare' la risposta di una data istanza. Per avere la possibilità di confrontare problemi diversi conviene tuttavia standardizzare la risposta dell'istanza. Si passa quindi a considerare una classe particolare di problemi che riveste un ruolo fondamentale nella teoria della complessità.

3.11 Definizione. *Un* problema di decisione *è un problema R in cui ogni stringa di Σ^* compare come stringa d'ingresso in esattamente una coppia di R e le uniche stringhe d'uscita sono 'sì' oppure 'no', includendo in 'no' anche l'uscita 'senza senso'.* ∎

Un problema di decisione R definisce quindi una partizione di Σ^* in due sottoinsiemi, il primo dei quali definito dalle stringhe d'ingresso associate alle stringhe d'uscita 'sì'. Si indichi con $L(R)$ questo sottoinsieme. Siccome un qualsiasi sottoinsieme L di Σ^* prende il nome di *linguaggio*, un problema di decisione R definisce quindi un linguaggio $L(R)$. Viceversa, dato un linguaggio L, si può definire il seguente problema di decisione

$$R(L) := \{(i, \text{'sì'}) : i \in L\} \cup \{(i, \text{'no'}) : i \notin L\} \ .$$

Si ha quindi una corrispondenza biunivoca naturale fra problemi di decisione e linguaggi. Risolvere un problema di decisione è perciò equivalente a decidere se una stringa appartiene al corrispondente linguaggio. In modo naturale possiamo invertire fra loro le uscite 'sì' e 'no' e ottenere un altro problema di decisione.

3.12 Definizione. *Si definisce* problema complementare *del problema di decisione R il problema di decisione $R(\Sigma^* - L(R))$ e si indica* co-R. ∎

A prima vista l'introduzione del problema complementare può sembrare un mero fatto formale. Infatti, se un algoritmo è in grado di decidere se $i \in L(R)$ oppure $i \notin L(R)$, il medesimo algoritmo automaticamente risolve anche il problema complementare. Tuttavia verrà introdotto nella sezione successiva un altro tipo di algoritmo, detto non deterministico, che opera in modo asimmetrico e per il quale un problema e il suo complementare sono essenzialmente diversi.

3.13 Esempio. Ad ogni grafo associamo la risposta 'sì' se è hamiltoniano o quella 'no' altrimenti. Un grafo può essere codificato in vari modi in una stringa di simboli. Dato che ogni codifica, purché 'ragionevole', può essere facilmente trasformata in un'altra, non è essenziale definire il tipo di codifica usata. Fissata una codifica, si associa alle stringhe che codificano un grafo hamiltoniano la risposta 'sì', a quelle che codificano un grafo non hamiltoniano la risposta 'no', e alle rimanenti stringhe che non codificano nessun grafo la

risposta 'no'. Il problema del circuito hamiltoniano può essere allora posto come il seguente problema di decisione: data una stringa, è la codifica di un grafo hamiltoniano?

Se si considera il problema complementare si deve associare la risposta 'sì' alle stringhe che codificano un grafo non hamiltoniano ed anche alle stringhe che non codificano nessun grafo, mentre la risposta 'no' è associata ai grafi hamiltoniani. Correttamente il problema complementare del circuito hamiltoniano dovrebbe essere: data una stringa, è vero che non corrisponde a nessun grafo oppure che codifica un grafo non hamiltoniano? In questa forma il problema ha un aspetto alquanto bizzarro, con questa mescolanza di stringhe valide e non valide. Si può però convenire una volta per tutte che sia facile decidere se una stringa codifica o no gli oggetti del problema in esame, per cui non si tiene nemmeno conto di questa possibile fase preliminare di verifica della validità della stringa e ci si concentra invece sull'aspetto fondamentale: è una codifica di tipo 'sì' oppure 'no'?

Adottata questa convenzione si può più ragionevolmente definire il problema del circuito hamiltoniano come: dato un grafo, è hamiltoniano? e il suo complementare come: dato un grafo, è vero che non è hamiltoniano? ∎

3.14 Esempio. Dato un problema di ottimizzazione (in forma di minimizzazione) si può facilmente associare ad esso un problema di decisione, aggiungendo all'istanza un valore K, e ponendo la domanda: dati f, F e K, esiste $x \in F$ tale che $f(x) < K$ (oppure $f(x) \leq K$)? Ovvero, esiste una soluzione ammissibile migliore (non peggiore) di un fissato valore K? Tale problema di decisione prende il nome di *versione ricognitiva* del problema di ottimizzazione.

Il problema complementare del problema di decisione appena definito è: dati f, F e K, è vero che $f(x) \geq K$ ($f(x) > K$) per ogni $x \in F$? In questo caso il problema complementare ha una particolare rilevanza pratica. Si supponga di conoscere una soluzione ammissibile y e di voler sapere se si tratta dell'ottimo. La domanda che ci si deve porre è: dati f, F e $y \in F$, è vero che $f(x) \geq f(y)$ per ogni $x \in F$? Si tratta quindi di un caso particolare di problema complementare della versione ricognitiva. ∎

3.15 Esempio. Dato un intero positivo si consideri il problema di decidere se è primo. Si tratta ovviamente di un problema di decisione e il suo complementare consiste nel decidere se è composto. ∎

Definito formalmente un problema di decisione, si può definire la complessità di un problema a partire dalla complessità degli algoritmi che lo risolvono. Più esattamente si definiscono classi di complessità alle quali diversi problemi possono appartenere o meno.

3.16 Definizione. *Si definisce classe* **P** *l'insieme dei problemi di decisione risolubili da un algoritmo polinomiale nel tempo.* ∎

3.17 Definizione. *Si definisce classe* **PSPACE** *l'insieme dei problemi di decisione risolubili da un algoritmo polinomiale nello spazio.* ∎

3.18 Definizione. *Si definisce classe* **L** *l'insieme dei problemi di decisione risolubili da un algoritmo logaritmico nello spazio.* ∎

La classe **P** è una delle più importanti. I problemi in essa contenuti sono da considerarsi 'facili' perché esistono algoritmi efficienti per la loro risoluzione. Andrebbe tenuto presente che si parla ora di problemi di decisione e non di problemi di ricerca, ma la cosa non riveste un aspetto essenziale. Se esiste un algoritmo polinomiale per la versione ricognitiva di un problema di ottimizzazione ne esiste anche uno per la versione originale (si rivedano l'esempio 1.78 e l'esercizio 1.79). e nell'altro senso la cosa è banalmente vera.

La classe **PSPACE** contiene praticamente tutti i problemi d'interesse. Solo problemi particolarmente intrattabili richiedono più di una quantità polinomiale di memoria. La classe **L** è contenuta in **P** per le considerazioni fatte nella sezione 3.2 sulla relazione fra risorse di tempo e di spazio usate da un algoritmo. Quindi problemi in **L** non solo sono facili, ma risultano anche parsimoniosi nell'uso dello spazio. Valgono quindi le inclusioni

$$\mathbf{L} \subset \mathbf{P} \subset \mathbf{PSPACE}\,.$$

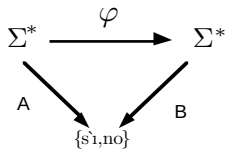
A tutt'oggi non si sa se le inclusioni $\mathbf{L} \subset \mathbf{P}$ e $\mathbf{P} \subset \mathbf{PSPACE}$ siano strette. Tuttavia è stato provato che $\mathbf{L} \neq \mathbf{PSPACE}$ e quindi almeno una delle due inclusioni deve essere stretta.

3.19 Esempio. Data una stringa di simboli di lunghezza n si decida se è un palindromo (un palindromo è una stringa che è uguale sia letta da sinistra che da destra). Un possibile algoritmo può verificare se i simboli in posizione 1 e n sono uguali, se lo sono passa a considerare quelli in posizione 2 e $n-1$ e così di seguito. Per operare in questo modo l'algoritmo deve mantenere in memoria interna i valori dei due puntatori e per fare questo basta una memoria $O(\log n)$. Quindi il problema del palindromo è in **L**. ∎

3.5. Trasformazioni e completezza

La teoria finora svolta permette di classificare alcuni problemi come facili se si è in grado di trovare un algoritmo polinomiale per risolverli. Che dire però quando, nonostante gli sforzi di moltissime persone, un determinato problema elude costantemente la possibilità di essere risolto in tempo polinomiale? Si può, con un certo atto di presunzione, affermare che è il problema ad essere troppo difficile? L'aspetto forse più affascinante della teoria della complessità computazionale consiste proprio nella possibilità di fare affermazioni rigorose sulla difficoltà di alcuni problemi, cioè sulla probabile impossibilità di risolverli in tempo polinomiale. Questo aspetto della teoria viene affrontato in questa e nella successiva sezione.

È ovviamente fondamentale poter confrontare le complessità di problemi diversi. A tal fine è utile il concetto di trasformazione di un problema in un altro. Considerato un problema di decisione come una funzione che mappa ogni stringa di Σ^* nella coppia $\{\text{'sì'}, \text{'no'}\}$, e assegnati due problemi di decisione A e B una trasformazione di A in B è una funzione φ che rende commutativo il seguente diagramma:



In altri termini ogni stringa che codifica un'istanza di A viene trasformata in una stringa che codifica un'istanza di B, e istanze di A di tipo 'sì' devono essere mappate in istanze di B di tipo 'sì'. Inoltre la funzione φ deve essere calcolabile in modo algoritmico. La complessità computazionale richiesta all'algoritmo di trasformazione dipende dalle classi di complessità di A e B. Se per esempio si è interessati a dimostrare che A è polinomiale a partire dal fatto che B è polinomiale, φ deve essere al più polinomiale. In questo modo A viene risolto, prima convertendo l'istanza di A in un'istanza di B, e poi risolvendo quest'ultima.

Per la commutatività del diagramma la risposta 'sì' o 'no' che si ottiene da B è la stessa di A. La complessità dell'algoritmo composto che si esegue è data dalla composizione delle due funzioni definenti le complessità della trasformazione e di B. Se queste due funzioni sono polinomi, la composizione di due polinomi è ancora un polinomio e quindi l'algoritmo composto per A è polinomiale. Si indichi con $A \leq_P B$ il fatto che A viene trasformato polinomialmente in B. In base a quanto esposto la relazione è transitiva, cioè $A \leq_P B$ e $B \leq_P C$ implicano $A \leq_P C$.

Se invece si vuole dimostrare che $A \in \mathbf{L}$ sapendo che $B \in \mathbf{L}$, una trasformazione φ polinomiale da A a B non è sufficiente, in quanto la composizione di φ con l'algoritmo di B è certamente polinomiale ma può richiedere più che spazio logaritmico. A questo fine bisogna imporre che anche la trasformazione richieda spazio logaritmico. Però non è così ovvio, come forse può sembrare al primo momento, che la composizione dei due algoritmi lavori con memoria interna logaritmica. Infatti le operazioni di scambio dati fra il primo e il secondo algoritmo, che risultano esterne per i due algoritmi, diventano interne per l'algoritmo composto. Si può tuttavia ottenere un algoritmo composto con le caratteristiche richieste facendo interagire in modo opportuno i due algoritmi. Si indichi con $A \leq_L B$ il fatto che A viene trasformato con spazio logaritmico in B. In base a quanto accennato la relazione è transitiva, cioè $A \leq_L B$ e $B \leq_L C$ implicano $A \leq_L C$.

Un'ulteriore classificazione di problemi all'interno di una classe è data dal concetto di *completezza*. Si supponga di sapere che tutti i problemi di **P** sono trasformabili tramite $\leq_L$ in un determinato problema A. Se si ottenesse per A un algoritmo logaritmico nello spazio, l'effetto di questo risultato sarebbe notevole, perché automaticamente la classe **P** collasserebbe in **L** a causa della transitività della trasformazione. Quindi è lecito considerare problemi A con questa caratteristica come i più rappresentativi, ovvero i più difficili, della classe, in quanto una loro risoluzione più efficiente si propaga su tutti gli altri problemi della classe. Analogamente si può ragionare per la classe **PSPACE** relativamente alle trasformazioni polinomiali. Abbiamo quindi le seguenti definizioni per i problemi più 'difficili' di ogni classe:

3.20 Definizione. *Si definisce* **P**-completo *ogni problema* $A \in \mathbf{P}$ *tale che* $R \leq_L A$ *per ogni* $R \in \mathbf{P}$. *La classe di problemi* **P**-*completi si indica con* **P-c**. ∎

3.21 Definizione. *Si definisce* **PSPACE**-completo *ogni problema* $A \in \mathbf{PSPACE}$ *tale che* $R \leq_P A$ *per ogni* $R \in \mathbf{PSPACE}$. *La classe di problemi* **PSPACE**-*completi si indica con* **PSPACE-c**. ∎

La notazione usata è quella universalmente adottata. Stranamente però le varie classi di problemi completi non hanno mai ricevuto una notazione definitiva. Di solito si usa la dizione, ad esempio, di 'problemi **P**-completi' e non si designa con un nome la relativa classe. A questo riguardo si fa notare che la notazione qui usata '**P-c**' e simili, per ovviare a tale mancanza, non è standard.

Le classi definite non sono vuote. Si può provare che il problema della programmazione lineare (in versione ricognitiva, se cioè un sistema di diseguaglianze lineari ammette soluzione) è P-completo e che il problema della soddisfattibilità quantificata (vedi esempio 1.81) è **PSPACE-c**. Si noti che se $\mathbf{L} \neq \mathbf{P}$ allora $\mathbf{P\text{-}c} \cap \mathbf{L} = \emptyset$. Similmente se $\mathbf{P} \neq \mathbf{PSPACE}$ allora $\mathbf{PSPACE\text{-}c} \cap \mathbf{P} = \emptyset$. In figura 3.1 viene rappresentata una 'mappa' delle varie classi finora introdotte nell'ipotesi $\mathbf{L} \neq \mathbf{P}$ e $\mathbf{P} \neq \mathbf{PSPACE}$. La transitività della relazione di trasformazione permette, una volta che sia noto almeno un problema A completo per una certa classe **C**, di dimostrare la completezza per la stessa classe di un problema X facendo vedere che:

1) $X \in \mathbf{C}$;
2) $A \leq X$.

Si vedranno più avanti diversi esempi di questa tecnica.

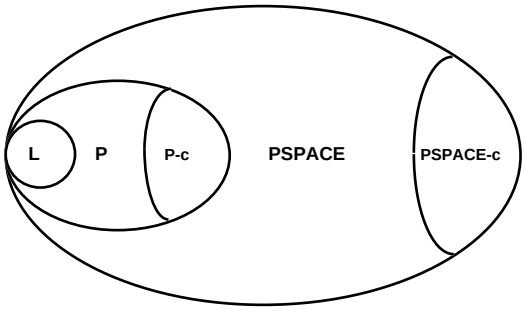


FIGURA 3.1

3.6. Algoritmi non deterministici e classe NP

I risultati precedenti fanno vedere come esistano problemi facili (quelli in **P**), alcuni dei quali probabilmente non ulteriormente migliorabili (quelli in **P-c**) e problemi molto probabilmente intrattabili (quelli in **PSPACE-c**). Rimane un'area molto ampia di problemi ugualmente classificati (quelli in **PSPACE**, ma non in **PSPACE-c** e non in **P**), fra i quali invece sarebbe importante poter discriminare ulteriormente.

Il concetto chiave per quest'analisi è quello di *algoritmo non deterministico*. In questo contesto il termine 'non deterministico' *non* significa 'stocastico', ma cerca invece di render conto della seguente modalità di calcolo: in un algoritmo normale, che ora per contrasto chiameremo *deterministico*, ogni operazione C è seguita da esattamente un'altra operazione $D(C)$ e il calcolo procede generando una sequenza di operazioni. In un algoritmo non deterministico invece può avvenire che a seguito di un'operazione C sia specificato un insieme $\mathcal{D}(C)$ di operazioni successive. Ciò che si intende per 'risolvere' in modo non deterministico un'istanza, limitatamente alle istanze di tipo 'sì', è che esista una sequenza di calcolo risolutiva in cui a C segue un'operazione $D \in \mathcal{D}(C)$. Un algoritmo non deterministico può essere rappresentato come un albero in cui ogni nodo C è un'operazione e i figli di ogni nodo sono le operazioni successive in $\mathcal{D}(C)$. Secondo questa rappresentazione un algoritmo deterministico è invece un semplice cammino. Allora 'risolvere' in modo non deterministico significa semplicemente che esista nell'albero un cammino dalla radice ad una foglia corrispondente ad una risoluzione effettiva, se l'istanza è di tipo 'sì'.

È più che mai opportuno rilevare che un algoritmo non deterministico è solo una costruzione mentale, utile ai fini teorici. In altre parole gli algoritmi non deterministici non esistono realmente e 'risolvere' in modo non deterministico non corrisponde affatto a risolvere nel senso usuale del termine. Un algoritmo non deterministico può essere simulato da uno deterministico, ma a prezzi molto elevati: si possono provare una alla volta tutte le operazioni di ogni insieme $\mathcal{D}$, cioè visitare l'albero finché non si trova la sequenza corretta di calcolo, e questo procedimento implica normalmente un tempo di calcolo esponenziale, oppure si può pensare che ad ogni nodo dell'albero il numero di processori si incrementi di quanto basta per eseguire contemporaneamente ed indipendentemente tutte le istruzioni successive, e quest'altro procedimento implica normalmente un numero di processori esponenziale.

C'è un altro modo di definire un algoritmo non deterministico che meglio si addice alla percezione intuitiva di 'calcolare' un problema ed anche permette più facilmente di determinare l'esistenza o meno di un algoritmo non deterministico polinomiale. Si pensi che il compito non sia tanto quello di risolvere un'istanza quanto di verificare che una soluzione, già pronta, sia corretta. Non importa come la soluzione sia stata ottenuta, si vuole soltanto raggiungere la convinzione che la soluzione è corretta. Inoltre si ha poco tempo da dedicare a questa verifica e quindi si vuole farla velocemente, ovvero in tempo polinomiale.

Un algoritmo non deterministico polinomiale corrisponde alla verifica appena esposta, cioè ad un algoritmo di verifica che, limitatamente alle istanze di tipo 'sì', sulla base dei dati del problema e di un opportuno insieme di dati aggiuntivi che chiameremo *certificato*, riesce a verificare in tempo polinomiale che l'istanza è proprio di tipo 'sì'. Affinché la verifica sia polinomiale è necessario che il certificato abbia lunghezza polinomiale altrimenti la sola lettura del certificato porterebbe via un tempo eccessivo. Certificati di lunghezza polinomiale vengono anche detti *succinti*.

Il legame fra questa verifica e la definizione precedente di algoritmo non deterministico consiste nel fatto che il certificato corrisponde ad individuare il cammino nell'albero che conduce alla soluzione. Per questo basta indicare, per ogni nodo C del cammino quale nodo in $\mathcal{D}(C)$ vada scelto come successivo. È importante notare che un algoritmo deterministico che simuli uno non deterministico polinomiale, anche se è esponenziale nel tempo, ha bisogno di uno spazio soltanto polinomiale, in quanto l'unica informazione necessaria in memoria è quella relativa al cammino dalla radice al nodo corrente.

3.22 Definizione. *Si definisce classe* **NP** *l'insieme dei problemi di decisione le cui istanze di tipo 'sì' sono verificabili in tempo polinomiale.* ∎

Per evitare possibili confusioni, vale la pena far presente che nella notazione '**NP**' la '**N**' sta per 'non deterministico' e non per 'non polinomiale'.

3.23 Esempio. Sia dato un grafo hamiltoniano. Come ci si può convincere che è davvero hamiltoniano? Basta che, insieme ai dati dell'istanza, venga fornita come certificato la sequenza dei nodi di un circuito hamiltoniano e poi verificare che i nodi sono tutti presenti senza ripetizioni e che fra due nodi successivi esiste un arco nel grafo. Questo calcolo ha complessità lineare e quindi il problema del circuito hamiltoniano è in **NP**.

Per questo problema si può facilmente pensare al seguente algoritmo non deterministico. Si prenda un nodo a caso e lo si ponga come radice dell'albero che rappresenta l'algoritmo non deterministico. Ogni nodo i dell'albero sia associato ad un nodo $j(i)$ del grafo e sia $J(i)$ l'insieme dei nodi del grafo associati ai nodi sull'albero del cammino dalla radice ad i (inclusi). Si assegni livello zero alla radice. Ogni nodo i dell'albero fino al livello $n-2$ (n numero dei nodi nel grafo) ha come immediati successori nodi associati a tutti i nodi del grafo adiacenti a $j(i)$ e non in $J(i)$ (se ve ne sono). Al livello $n-1$ vi può essere fra i successori di i il nodo iniziale se adiacente a $j(i)$.

L'istanza è di tipo 'sì' se almeno una delle foglie dell'albero è associata al nodo iniziale e il cammino dalla radice alla foglia identifica un circuito hamiltoniano. Necessariamente tale foglia si trova al livello n. Si veda in figura 3.2(a) un grafo hamiltoniano e in figura 3.2(b) il relativo albero dell'algoritmo non deterministico. Le quattro foglie a livello 5 corrispondono ai 4 circuiti hamiltoniani (ogni circuito appare in realtà due volte perché può essere percorso in versi opposti). ∎

3.24 Esempio. Il problema della soddisfattibilità è un problema di decisione che sta in **NP** in quanto basta fornire l'assegnamento che rende vera la formula per verificare la correttezza della soluzione. ∎

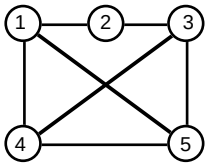


FIGURA 3.2(A)

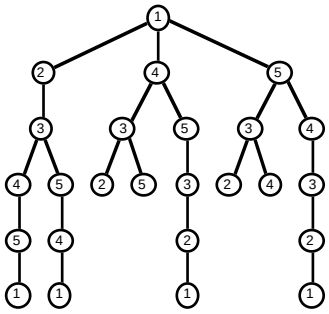


FIGURA 3.2(B)

3.25 ESEMPIO. Verificare che un'istanza del problema della soddisfattibilità quantificata sia di tipo 'sì' si può fare assegnando ad ogni nodo dell'albero degli assegnamenti (vedi esempio 1.81) l'opportuno assegnamento per la variabile corrispondente. C'è un problema però. Un certificato del genere è di grandezza esponenziale rispetto al numero di variabili. Infatti ciò che viene richiesto è una strategia completa sulle decisioni da prendere in ogni possibile circostanza. Ci sono valide ragioni per credere che un certificato polinomiale non possa esistere. Si tratta infatti di un problema **PSPACE**-completo e quindi è plausibile che il problema della soddisfattibilità quantificata non appartenga ad **NP**. ∎

3.26 ESEMPIO. Si consideri un sistema di diseguaglianze lineari $A\,x \leq b$. Ci si chiede se esiste una soluzione ammissibile intera. Se una tale soluzione esiste un certificato è dato dalla soluzione stessa e la verifica è poi banale. Tuttavia la cosa è più complessa di quanto sembri a prima vista. Che garanzia c'è che la soluzione non si esprima con un numero di cifre esponenziale rispetto ai dati del problema? Ad esempio per il seguente sistema:

$$\begin{aligned}(a\,b-1)\,x_1 - b\,x_2 + 1 &\geq 0\\ (a\,b+1)\,x_1 - b\,x_2 - 1 &\leq 0\\ b\,x_1 + b\,x_2 - 1 &\geq 0\end{aligned}$$

con a e b interi positivi, l'unica soluzione intera è $(1, a)$. Il valore a potrebbe essere molto grande. Però in questo caso anche la descrizione dell'istanza sarebbe altrettanto grande. Si può dimostrare che, se esiste una soluzione intera per un sistema di disequazioni, allora la soluzione si codifica con un numero polinomiale di simboli rispetto alla descrizione dell'istanza. Quindi il certificato dato dalla soluzione stessa è succinto ed il problema della programmazione lineare intera (versione ricognitiva) sta in **NP**. ∎

La definizione di algoritmo non deterministico è asimmetrica. Si pone l'accento solo sulle istanze di tipo 'sì' e non si chiede nulla a quelle di tipo 'no'. È ovvio quindi che, dal punto di vista di un algoritmo non deterministico, un problema e il suo complementare sono problemi diversi. Se prendiamo un problema in **NP** e consideriamo il suo complementare non è affatto detto che anche il complementare sia in **NP**. Anzi, per un'ampia classe di problemi ci sono buone ragioni per credere che non appartengano entrambi a **NP**.

3.27 ESEMPIO. Si consideri il complemento del problema del circuito hamiltoniano: dato un grafo, è vero che non è hamiltoniano? A tutt'oggi nessuno è riuscito a fornire un certificato succinto che dimostri la non esistenza di circuiti hamiltoniani. Un elenco completo di tutti i circuiti e una verifica che nessuno di essi contiene tutti i nodi non è certamente succinto. ∎

È naturale, a questo punto, definire un'altra classe di problemi, complementare ad **NP**.

3.28 Definizione. *Si definisce classe* **co-NP** *l'insieme dei problemi di decisione le cui istanze di tipo 'no' sono verificabili in tempo polinomiale.* ∎

Ogni problema in **NP** ha un complementare che, per definizione, sta in **co-NP**. Ecco altri due esempi di problemi in **co-NP**.

3.29 Esempio. (vedi esempio 3.15) Dato un intero, è primo? Se non è primo, basta fornire una fattorizzazione. Quindi il problema della primalità sta in **co-NP**. ∎

3.30 Esempio. Sia data una formula booleana in forma congiuntiva. Si vuole sapere se è valida, cioè se è vera per qualsiasi valore delle variabili. Se la formula non è valida, basta fornire un assegnamento che rende falsa la formula. Quindi il problema della validità sta in **co-NP**. Si noti che è il complemento di una variante del problema della soddisfattibilità, in cui la formula booleana si presenta con una negazione e quindi è una formula in forma disgiuntiva, che poi mediante le regole formali della logica si trasforma in un forma congiuntiva. ∎

3.31 Esempio. (vedi esempio 3.14) Dato un problema di ottimizzazione, e una soluzione ammissibile y, vogliamo sapere se y è ottimo. Se y non è ottimo basta fornire una soluzione z migliore di y. ∎

Tutti i problemi in **P** appartengono naturalmente sia a **NP** che a **co-NP**. Infatti la risoluzione stessa, essendo polinomiale, costituisce una verifica valida sia per le istanze di tipo 'sì' che per quelle di tipo 'no'. Inoltre, per le considerazioni fatte precedentemente sulla simulazione di un algoritmo non deterministico da parte di uno deterministico, sia **NP** che **co-NP** sono contenute in **PSPACE**. Allora valgono le seguenti inclusioni:

$$\mathbf{P} \subset \mathbf{NP} \subset \mathbf{PSPACE} \qquad \mathbf{P} \subset \mathbf{co\text{-}NP} \subset \mathbf{PSPACE} \tag{3.3}$$

Le domande fondamentali riguardano la validità delle seguenti eguaglianze

$$\mathbf{P} = \mathbf{NP} \ ?$$

(che implicherebbe $\mathbf{NP} = \mathbf{co\text{-}NP}$),

$$\mathbf{NP} = \mathbf{PSPACE} \ ?$$

(che implicherebbe $\mathbf{NP} = \mathbf{co\text{-}NP}$), e

$$\mathbf{NP} = \mathbf{co\text{-}NP} \ ?$$

(che invece non implicherebbe $\mathbf{P} = \mathbf{NP}$). Le tre questioni sono a tutt'oggi irrisolte. Tuttavia si pensa, sulla base dei risultati parziali ottenuti in tutti questi anni, che le inclusioni (3.3) siano strette e che $\mathbf{NP} \neq \mathbf{co\text{-}NP}$.

Delle tre questioni, la più intrigante sembra essere la prima. Sarebbe un fatto davvero sorprendente se la risoluzione di un problema fosse altrettanto facile quanto la sua verifica, e il buon senso tende quindi a scartare la possibilità di avere $\mathbf{P} = \mathbf{NP}$. Un modo per dimostrare la stretta inclusione consisterebbe nel trovare un problema in **NP** di cui si possa anche provare l'impossibilità di una risoluzione polinomiale. Ma non si vede come si possa provare una tale impossibilità. La strada per provare la congettura opposta, che cioè $\mathbf{P} = \mathbf{NP}$, sarebbe invece facilmente delineata, andando alla ricerca di un algoritmo polinomiale per almeno un problema **NP**-completo (vedi sotto), tanto che vi sono stati molti tentativi di ricerca, tutti falliti. Considerato l'enorme sforzo investito in questi venticinque anni senza risolvere la congettura, sono in molti a pensare che probabilmente nel problema si nasconde qualcosa di molto più profondo, che attualmente non si riesce a vedere. La situazione può ricordare il millenario problema di dimostrare il quinto postulato di Euclide a partire

dai primi quattro, problema alla fine abbandonato con l'introduzione delle geometrie non euclidee, cioè con un mutamento radicale nella concezione della geometria. Si tratta forse di qualcosa di simile anche per $\mathbf{P} = \mathbf{NP}$?

Si può applicare il concetto di non determinismo anche alle classi **L** e **PSPACE**. Riguardo alla prima classe si ha:

3.32 Definizione. *Si definisce classe* **NL** *l'insieme dei problemi di decisione le cui istanze di tipo 'sì' sono verificabili in spazio logaritmico.* ∎

Riguardo alla seconda classe, si è visto che il non determinismo non apporta un vantaggio dal punto di vista dello spazio, per cui i problemi verificabili in spazio polinomiale coincidono con quelli risolubili in spazio polinomiale. Quindi non c'è bisogno di definire una nuova classe di problemi. Si può dimostrare che valgono le inclusioni (senza che si sappia se sono strette):

$$\mathbf{L} \subset \mathbf{NL} \subset \mathbf{P}$$

Anche per le classi definite da un algoritmo non deterministico si può applicare il concetto di completezza. Quindi si ha:

3.33 Definizione. *Si definisce problema* **NP***-completo ogni problema* $A \in \mathbf{NP}$ *tale che* $R \leq_P A$ *per ogni* $R \in \mathbf{NP}$*. La classe di problemi* **NP***-completi si indica con* **NP-c**. ∎

La nozione di completezza è particolarmente importante per la classe **NP**. I problemi **NP**-completi sono verosimilmente non polinomiali, altrimenti **NP** collasserebbe in **P**, e quindi si tratta di problemi la cui difficoltà è 'garantita'. Inoltre, ammesso sia vero che **NP** è strettamente contenuto in **PSPACE**, i problemi **NP**-completi sono meno intrattabili dei corrispettivi **PSPACE**-completi e come tali risultano più comuni dal punto di vista dell'ottimizzazione. Per questi motivi il concetto di **NP**-completezza è fondamentale nello studio della complessità dei problemi di ottimizzazione. La classe dei problemi **NP**-completi non è vuota. Infatti si ha il seguente fondamentale teorema:

3.34 Teorema. (S. Cook [1971]) *Il problema della soddisfattibilità è* **NP***-completo.* ∎

Per una dimostrazione del teorema di Cook si può vedere, oltre all'articolo originale, Garey e Johnson [1979] p. 39-44, Papadimitriou [1994] p. 171-172, oppure (in una versione molto compatta) W.J. Cook et al. [1998] p. 317-318.

Una volta trovato un problema **NP**-completo, non serve dimostrare la trasformabilità di ogni problema in **NP** ad un particolare problema X di cui si vuol scoprire la complessità, basta, come già detto, dimostrare che $X \in \mathbf{NP}$ e poi che $A \leq_P X$, per un opportuno problema A di cui è nota l'appartenenza a **NP-c**. La parte difficile è la trasformazione. Bisogna scegliere il problema A in modo che la trasformazione risulti naturale, ma anche così è normalmente richiesta una buone dose di 'arte' per operare la trasformazione. Si vedranno diversi esempi nella prossima sezione.

Una definizione simile di completezza vale per i problemi in **co-NP**.

3.35 Definizione. *Si definisce problema* **co-NP***-completo ogni problema* $A \in \mathbf{co\text{-}NP}$ *tale che* $R \leq_P A$ *per ogni* $R \in \mathbf{co\text{-}NP}$*. La classe di problemi* **co-NP***-completi si indica con* **co-NP-c**. ∎

Un'ovvia caratterizzazione dei problemi **co-NP**-completi è data dal seguente risultato:

3.36 Teorema. *Un problema è* **NP***-completo se e solo se il suo complementare è* **co-NP***-completo.* ∎

Quindi, per quanto visto precedentemente, il problema della validità è **co-NP**-completo. Se $\mathbf{NP\text{-}c} \cap \mathbf{co\text{-}NP\text{-}c} \neq \emptyset$ allora ogni problema in **NP** avrebbe un certificato polinomiale anche per le istanze 'no' (tramite la trasformazione in **NP** al problema in comune e il fatto che questo sta in **co-NP**) e similmente ogni problema in **co-NP** avrebbe un certificato polinomiale per le istanze 'sì', quanto a dire che $\mathbf{NP} = \mathbf{co\text{-}NP}$. Essendo questa uguaglianza altamente improbabile si è portati a credere che $\mathbf{NP\text{-}c} \cap \mathbf{co\text{-}NP\text{-}c} = \emptyset$.

Per valutare meglio l'implicazione teorica di questo risultato ci si chieda ad esempio se possa esistere un teorema del tipo: 'Un grafo possiede un circuito hamiltoniano se e solo se una certa condizione è soddisfatta'. Sappiamo che un simile teorema esiste per i circuiti euleriani. C'è la speranza che qualcuno scopra una condizione necessaria e sufficiente anche per i circuiti hamiltoniani? A scanso di equivoci diciamo subito che la condizione deve essere succinta, qualcosa cioè che si verifichi senza eccessivo sforzo computazionale. Un teorema che richiedesse un tempo esponenziale di calcolo per la verifica della condizione non avrebbe molto senso. Allora una tale condizione diventerebbe un certificato succinto sia per le istanze 'sì' che per quelle 'no', e abbiamo appena visto come tale circostanza sia da ritenersi improbabile. Analogamente sarebbe una fatica forse vana cercare dei teoremi generali del tipo 'la soluzione $\hat{x}$ è ottima se e solo se ecc.'. Se un tale teorema deve intendersi anche per problemi **NP**-completi la sua esistenza è improbabile. Vedremo che esistono condizioni necessarie e sufficienti ma per problemi polinomiali e anche che esistono condizioni generali solo sufficienti o solo necessarie, ma non ambedue.

Una definizione di completezza può essere data anche per la classe **NL**.

3.37 Definizione. *Si definisce problema* **NL***-completo ogni problema* $A \in \mathbf{NL}$ *tale che* $R \leq_L A$ *per ogni* $R \in \mathbf{NL}$. *La classe di problemi* **NL***-completi si indica con* **NL-c**. ∎

3.38 Esempio. Sia dato un grafo e siano assegnati due nodi particolari s e t. Tali nodi sono connessi? Si può dimostrare che questo problema di decisione è **NL**-completo. ∎

3.39 Esercizio. Si dimostri che $\mathbf{NL} = \mathbf{co\text{-}NL}$. ∎

La mappa rivista delle varie classi di problemi, nelle ipotesi più verosimili di inclusioni strette risulta essere quella di figura 3.3.

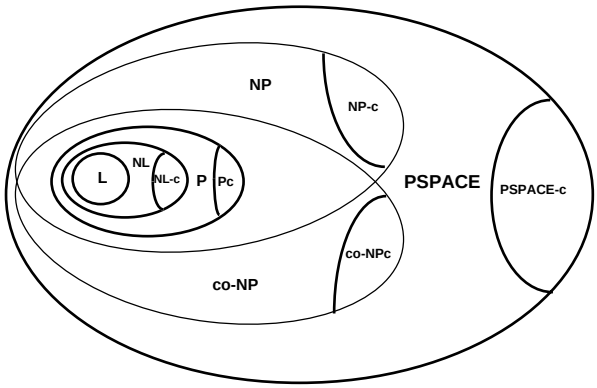


Figura 3.3

Vi sono problemi che, pur essendo **NP**-completi, posono essere risolti con un algoritmo pseudopolinomiale. Questa situazione viene vista come favorevole dal punto di vista della complessità. Quindi, per discriminare ulteriormente questi problemi **NP**-completi dagli altri, che verosimilmente non possono essere risolti da un algoritmo pseudopolinomiale si ha le seguente:

3.40 DEFINIZIONE. *I problemi* **NP***-completi che rimangono tali, anche adottando la codifica unaria per i numeri della stringa d'ingresso, prendono il nome di* fortemente **NP**-completi. ∎

Un algoritmo pseudopolinomiale per un problema fortemente **NP**-completo sarebbe di fatto un algoritmo polinomiale e quindi l'esistenza di un algoritmo pseudopolinomiale per un problema fortemente **NP**-completo significherebbe $\mathbf{P} = \mathbf{NP}$. Quindi i problemi fortemente **NP**-completi vanno visti come più intrattabili degli altri. Il problema della soddisfattibilità è fortemente **NP**-completo in quanto, banalmente, non compaiono numeri nelle sue stringhe d'ingresso. Per dimostrare la forte **NP**-completezza di un problema X in cui compaiono numeri nella stringa d'ingresso bisogna che la trasformazione $A \leq_P X$ da un problema A sia tale che: 1) A è fortemente **NP**-completo, 2) la trasformazione è polinomiale rispetto alla codifica unaria.

Per quel che riguarda invece problemi in **NP** ma non **NP**-completi, risultano particolarmente interessanti alcuni problemi che si situano in una posizione intermedia fra quelli polinomiali e quelli **NP**-completi. Sono problemi di cui si è provata l'appartenenza a $\mathbf{NP} \cap \mathbf{co\text{-}NP}$ e per i quali non si è ancora trovato un algoritmo polinomiale. Siccome appare molto dubbio che **NP** possa coincidere con **co-NP**, è improbabile che tali problemi possano essere o **NP**-completi o **co-NP**-completi. Quindi sono problemi più trattabili e la teoria non esclude la possibilità di trovare un algoritmo polinomiale per questi.

3.41 ESEMPIO. Il problema di decidere se un numero è primo appartiene a **co-NP** in modo banale, come si è già visto. Un teorema di teoria dei numeri afferma che un numero p è primo se e solo se esiste un numero $1 < r < p$ tale che $r^{p-1} = 1 \bmod p$, ed inoltre $r^{(p-1)/q} \neq 1 \bmod p$ per tutti i divisori primi q di $p-1$. La dimostrazione che r e i divisori primi di $p-1$ costituiscono un certificato polinomiale è alquanto laboriosa e il lettore è rinviato a Pratt [1975] oppure a Papadimitriou [1994] p. 222.

Come si sa, la fattorizzazione di numeri interi con moltissime cifre sta alla base dei moderni metodi di crittografia. Questi metodi sfruttano il fatto che non esistono algoritmi efficienti per fattorizzare numeri interi ottenuti come prodotto di due primi. Però, per quanto detto, la teoria non esclude l'esistenza di algoritmi polinomiali e forse questa è già un fatto, ma coperto dal segreto. ∎

3.42 ESEMPIO. Il problema della programmazione lineare (in forma ricognitiva) si trovava fino al 1979 in una situazione simile al problema della primalità. Non era ancora noto un algoritmo polinomiale, però sia le istanze di tipo 'sì' che quelle di tipo 'no' avevano un certificato. Questo tipo di certificato si può applicare a tutti i problemi di ottimizzazione per cui, in base alla teoria della dualità (vedi il capitolo 5) il valore ottimo (del problema di minimizzazione) è anche il valore ottimo di un altro problema (di massimizzazione), detto duale. Si consideri il problema di decisione come formulato nell'esempio 3.14: se l'istanza è di tipo 'sì' basta banalmente trovare x ammissibile tale che $f(x) < K$. Se l'istanza è di tipo 'no' vuol dire che $f(x) \geq K$ per ogni x ammissibile, e quindi $v \geq K$. Siccome v è il valore massimo di un altro problema, basta fornire una soluzione dell'altro problema con valore $\geq K$.

Questo stato particolare della programmazione lineare faceva sospettare che fosse possibile un algoritmo polinomiale e questo fu effettivamente scoperto da Khachiyan [1979]. ∎

3.43 Esempio. Questo esempio è un caso particolare dell'esempio precedente. È noto che in un grafo bipartito la cardinalità di un accoppiamento massimo è uguale alla cardinalità di una minima copertura di nodi (si veda l'esercizio 1.60). Decidere se esiste un accoppiamento di cardinalità maggiore di K è un problema polinomiale e quindi sta ovviamente in **NP** ∩ **co-NP**. Tuttavia è interessante far vedere i certificati che in questo caso sono più semplici della risoluzione del problema. Il certificato per le istanze di tipo 'sì' è ovvio e per quelle di tipo 'no' si basa sulla proprietà citata (che alla fine è di dualità). Basta quindi fornire un insieme di nodi che sia una copertura (verifica immediata) e che sia di cardinalità al più K (esistenza garantita dalla proprietà). È opportuno far notare ancora una volta che non si chiede di 'trovare' il certificato, ma solo di verificarlo e non è appunto compito del verificatore trovare i nodi della copertura. ∎

Vi sono inoltre problemi in **NP** che però non sono in **P**, né in **NP-c**, e neppure in **co-NP** (o, meglio, non si è provata l'appartenenza a nessuna di queste classi). Si noti che si è provato (Ladner [1975]) che **P** e **NP-c** non possono formare una partizione di **NP**. Quindi o **P**=**NP** oppure devono esistere problemi non polinomiali e neppure **NP**-completi. Per il momento l'importante problema di determinare se due grafi sono isoformi appartiene a questa categoria.

3.7. Problemi NP-completi

In questa sezione vengono presentati alcuni problemi **NP**-completi insieme con le dimostrazioni della loro **NP**-completezza. È molto importante impadronirsi delle tecniche di dimostrazione delle trasformazioni. Normalmente dimostrare la **NP**-completezza di un problema richiede molta fantasia e un'abilità che si acquista solo con una certa esperienza della materia. Anche se l'elenco dei problemi **NP**-completi ne comprende ormai migliaia e probabilmente un problema con cui si deve lavorare è già incluso nell'elenco, pur tuttavia avviene spesso che si deve affrontare un problema nuovo e c'è la necessità di dimostrare almeno che sia **NP**-completo, se non si è in grado di trovare un algoritmo polinomiale. Inoltre vi sono ancora in letteratura problemi 'aperti', per i quali cioè non si è né trovato un algoritmo polinomiale né dimostrata la **NP**-completezza. Quindi è utile avere una certa pratica nel trasformare problemi.

L'articolo fondamentale che iniziò la 'caccia' ai problemi **NP**-completi si deve a Karp [1972]. Successivamente, una volta aperta la strada, decine e decine di problemi si aggiunsero all'elenco. In Garey e Johnson [1979] è riportato un elenco con più di 300 problemi. Un elenco aggiornato probabilmente conterrebbe migliaia di nomi e scoprire se di un problema è già stata dimostrata la **NP**-completezza costituisce di per sé un problema!

3.44 Esempio. Dimostriamo che il problema di decidere se il seguente insieme

$$\begin{aligned} A\,x &\geq b \\ x &\in \{0,1\}^n \end{aligned} \tag{3.4}$$

sia ammissibile è **NP**-completo. Si tratta quindi della versione ricognitiva della programmazione lineare 0-1. La dimostrazione di appartenenza a **NP** è semplice, perché una qualsiasi soluzione ammissibile costituisce un certificato succinto. Per dimostrare la completezza si opera una trasformazione dal problema della soddisfattibilità. Quindi bisogna trasformare una generica istanza della soddisfattibilità in una particolare istanza di (3.4) in modo però che istanze 'sì' del primo problema si trasformino in istanze 'sì' del secondo e altrettanto per le istanze 'no'.

A questo scopo si associ ad ogni variabile logica della formula una variabile 0-1 di (3.4) con l'idea che ad un assegnamento di vero alla variabile logica corrisponde il valore 1 della variabile 0-1 e ad ogni clausola della formula una diseguaglianza di (3.4). Si consideri una particolare clausola. La clausola viene trasformata in una somma di tanti termini quanti sono i letterali nella clausola. Se nella clausola compare il letterale x_i, nella somma compare la variabile 0-1 corrispondente x_i. Se invece compare il letterale negato $\neg x_i$, nella somma compare il termine $(1 - x_i)$. Siccome basta che uno solo dei letterali sia vero, la somma deve essere vincolata ad essere ≥ 1. Ad esempio

$$(x_1 \vee x_2 \vee \neg x_3) \quad \Longrightarrow \quad x_1 + x_2 + (1 - x_3)$$

e si deve dimostrare che

$$(x_1 \vee x_2 \vee \neg x_3) = V \quad \Longleftrightarrow \quad x_1 + x_2 + (1 - x_3) \geq 1 \tag{3.5}$$

A questo fine basta considerare un'istanza 'sì' del primo termine in (3.5) e far vedere che soddisfa il secondo termine e poi considerare un'istanza 'sì' del secondo termine in (3.5) e far vedere che soddisfa il primo termine. Si tratta in questo caso di una verifica molto semplice che viene lasciata al lettore. Siccome nel problema della soddisfattibilità ogni clausola deve essere soddisfatta, ogni somma derivata da ogni clausola deve essere soddisfatta e quindi si ha (3.4). La trasformazione ha complessità lineare e quindi è valida. La trasformazione rimane di complessità lineare anche con la codifica unaria e quindi la programmazione lineare 0-1 è fortemente **NP**-completa. Ad esempio

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \tag{3.6}$$

viene trasformata in

$$\begin{aligned} x_1 + x_2 - x_3 \quad &\geq \quad 0 \\ -x_1 + x_2 \quad - x_4 &\geq -1 \\ -x_1 \quad - x_3 + x_4 &\geq -1 \\ - x_2 + x_3 - x_4 &\geq -1 \\ x_i \in \{0,1\} \ \forall i & \end{aligned}$$

Come si vede, disponendo di un algoritmo che risolve (3.4) si è in grado di risolvere ogni istanza della soddisfattibilità. ∎

3.45 Esempio. Si è accennato nel capitolo 1 al fatto che massimizzare una funzione convessa su un insieme definito da diseguaglianze lineari è difficile. Si vuole appunto far vedere che la versione ricognitiva è **NP**-completa. Il problema si può porre come: data $f : R^n \to R$ convessa, $F := \{x \in R^n : A\,x \leq b\}$ e K, esiste $x \in F$ tale che $f(x) \geq K$?

La trasformazione avviene dalla programmazione lineare 0-1 (in versione ricognitiva), cioè: esiste $\hat{x} \in \{0,1\}^n$ tale che $A\,\hat{x} \leq b$? Basta definire la funzione convessa

$$f(x) := \sum_{i=1}^{n} x_i\,(x_i - 1)$$

e aggiungere a $A\,x \leq b$ il vincolo $0 \leq x_i \leq 1$, $\forall i$. Si noti che $f(x) \leq 0$ su $[0,1]^n$ e $f(x) = 0$ in $[0,1]^n$ se e solo se $x \in \{0,1\}^n$. Quindi basta porre la domanda: esiste $\hat{x}$ tale che $A\,\hat{x} \leq b$, $0 \leq \hat{x}_i \leq 1$ e $f(\hat{x}) \geq 0$? È immediato verificare che la trasformazione è polinomiale e che c'è corrispondenza fra istanze di tipo 'sì' e tipo 'no'. La proprietà di appartenere a **NP** richiede lo stesso tipo di ragionamento fatto per la programmazione lineare intera. ∎

3.46 Esempio. Questo esempio di trasformazione è un po' più complesso. Si vuole dimostrare che il problema del massimo insieme stabile (in forma ricognitiva) è **NP**-completo. La trasformazione scelta è ancora dalla soddisfattibilità.

Per cominciare si può osservare che nella soddisfattibilità ci sono delle clausole (siano m) e si vuole che siano tutte soddisfatte. Un'idea potrebbe essere quella di far corrispondere i nodi dell'insieme stabile (nodi stabili) alle clausole: tante clausole soddisfatte, altrettanti nodi stabili. Quindi i nodi stabili dovranno essere almeno tanti quante le clausole se la formula è soddisfattibile. Raffinando quest'idea si può pensare che il nodo stabile che rappresenta la clausola corrisponda ad uno dei letterali veri della clausola. Affinché quest'idea funzioni si deve, 1) associare ad ogni letterale un nodo del grafo, 2) avere al più un nodo stabile per ogni insieme associato ad una clausola, 3) non avere nodi stabili in corrispondenza di un letterale e della sua negazione.

Per ottenere 2) basta collegare in cricca tutti i nodi associati alla clausola e per ottenere 3) basta collegare ogni nodo di un letterale con il nodo del corrispondente letterale negato. Ad esempio la formula (3.6), le cui clausole indichiamo con A, B, C e D, genera il grafo in figura 3.4, che ha quattro cricche di tre nodi (la clausola è indicata all'interno delle cricche di tre nodi, i nodi bianchi si riferiscono a letterali normali, quelli neri a letterali negati, i numeri dentro i nodi alle variabili).

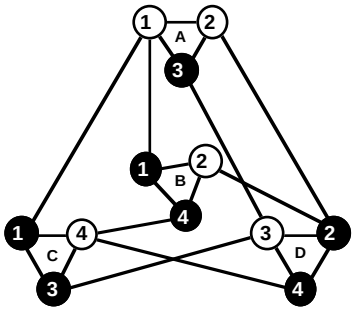


Figura 3.4

Costruito il grafo bisogna ora dimostrare la corrispondenza fra le istanze di tipo 'sì'. Nel problema di decisione dell'insieme stabile il numero di nodi stabili deve essere almeno pari al numero di clausole. Si supponga sapprima che la formula booleana sia soddisfatta. Bisogna far vedere che esistono m nodi stabili. Se la formula è soddisfatta, in ogni clausola c'è almeno un letterale vero. Si prenda a caso un letterale vero da ogni formula e si considerino i nodi corrispondenti. Dobbiamo far vedere che fra questi nodi non ci sono archi. Infatti non ci possono essere gli archi delle cricche perché ogni nodo proviene da una cricca diversa. Gli altri archi collegano (i nodi di) un letterale con la sua negazione e non può essere che nella formula soddisfatta siano contemporaneamente veri un letterale e la sua negazione. Quindi i nodi scelti sono davvero un insieme stabile di cardinalità m. Si è così dimostrato che ogni istanza 'sì' della soddisfattibilità si trasforma in un'istanza 'sì' dell'insieme stabile.

Bisogna ora dimostrare la stessa cosa per le istanze 'no'. È equivalente dimostrare che ogni istanza 'sì' dell'insieme stabile si trasforma in un'istanza 'sì' della soddisfattibilità. Si supponga quindi che esistano almeno m nodi stabili. Innanzitutto dimostriamo che non possono essere più di m e questo lo si ottiene dal fatto che i nodi sono partizionati in m cricche. Quindi ci sono esattamente m nodi stabili e necessariamente uno per cricca. Se al nodo stabile è associato un letterale normale la variabile corrispondente viene dichiarata vera, altrimenti viene dichiarata falsa. Non potrà mai essere che una stessa variabile sia contemporaneamente dichiarata vera e falsa a causa degli archi che collegano i letterali con

le proprie negazioni. Non è detto che in questo modo si assegni un valore di verità a tutte le variabili. Quelle che non sono associate ad alcun nodo stabile possono ricevere un arbitrario valore di verità. Comunque l'assegnamento fatto con i nodi stabili è già sufficiente a rendere soddisfatta la formula.

Il problema dell'insieme stabile è banalmente in **NP**. Quindi è **NP**-completo. Automaticamente si è dimostrata anche la **NP**-completezza della cricca massima, della minima copertura nodi, della minima copertura d'insiemi e del massimo impaccamento d'insiemi (si rivedano gli esercizi 1.58, 1.66 e l'esempio 1.68). Tutti questi problemi sono inoltre fortemente **NP**-completi dato che gli unici numeri che compaiono nelle descrizioni delle istanze (la cardinalità dei sottoinsiemi con le proprietà richieste) sono limitate linearmente rispetto alla rimanente parte della stringa d'ingresso (ad esempio numero di nodi). ∎

3.47 Esempio. Questo esempio è di difficoltà superiore al precedente e ben rappresenta il livello di ingegnosità richiesto in alcuni casi. Il problema di cui si vuole dimostrare la **NP**-completezza è il circuito hamiltoniano. Varie dimostrazioni si trovano in letteratura con trasformazioni dalla copertura di nodi, di cui si è appena dimostrata la **NP**-completezza, (vedi Garey e Johnson [1979] oppure Maffioli [1990]), oppure da una variante della soddisfattibilità chiamata 3-soddisfattibilità, in cui le clausole hanno esattamente tre letterali (Papadimitriou [1995]). Qui verrà scelta la soddisfattibilità adattando di poco la costruzione in Papadimitriou [1995]. Possiamo escludere il caso di clausole con un solo letterale, perché in questo caso il letterale deve essere banalmente vero e quindi la formula si semplifica di conseguenza. Si noti subito che, una volta dimostrata la **NP**-completezza, è automaticamente dimostrata la forte **NP**-completezza in quanto non compaiono numeri nelle istanze del circuito hamiltoniano.

Si cominci con il notare che, in una clausola con p letterali, $2^p - 1$ assegnazioni di valori di verità rendono la clausola soddisfatta ed uno solo no. L'idea chiave di tutta la trasformazione risiede nel trovare una simile proprietà relativamente ai circuiti di un grafo. Si considerino p nodi collegati in circuito (sia C questo circuito) all'interno di un grafo e un possibile circuito hamiltoniano H che, prima o dopo, deve attraversare questi nodi (se $p = 2$ si crea un circuito con due archi paralleli). Il circuito hamiltoniano H può attraversare o meno gli archi di C. Solo una modalità di attraversamento è esclusa a priori, cioè quella che passa per tutti gli archi di C (se vi sono altri nodi del grafo oltre ai p nodi). Tutti gli altri $2^p - 1$ attraversamenti sono a priori ammissibili. Si vedano in figura 3.5 i sette modi in cui un circuito hamiltoniano può attraversare un circuito di $p = 3$ nodi; l'ottavo modo è quello non ammissibile per un circuito hamiltoniano. Un tale circuito C venga detto *circuito di clausola*. L'idea allora è di associare ogni attraversamento di un arco di clausola con il valore falso del corrispondente letterale.

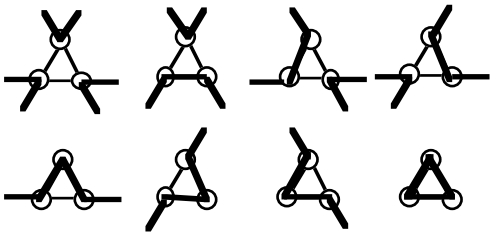


Figura 3.5

Un problema da risolvere è quello di obbligare tutti i letterali associati ad una medesima variabile ad assumere valori di verità coerenti. In primo luogo si costruiscano nodi e archi che

servano ad assegnare valori di verità alle variabili. In figura 3.6 si vede un tale dispositivo: vi sono $n+1$ nodi (nodi di assegnamento, n è il numero di variabili booleane). Ogni nodo è collegato al successivo con due archi paralleli (archi di assegnamento). Gli archi che collegano il nodo i con il nodo $i+1$ sono rispettivamente associati al valore vero o falso della variabile x_i. L'idea è che la scelta operata dal circuito hamiltoniano su quale arco usare nel passare da un nodo al successivo corrisponda alla scelta del valore di verità della variabile.

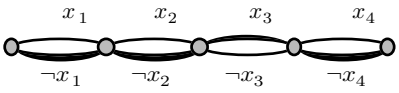


FIGURA 3.6

Ora si tratta di obbligare ogni letterale ad assumere il valore imposto nella scelta fatta quando il circuito hamiltoniano passa per gli archi di assegnamento. Allo scopo serve il dispositivo illustrato in figura 3.7. In figura 3.7.a sono raffigurati due archi, uno di assegnamento (nodi grigi) e l'altro di clausola (nodi neri). Si supponga di voler imporre la condizione che esattamente uno dei due archi sia attraversato da un circuito (usando anche i nodi estremi degli archi). Si aggiungano agli archi i nodi (bianchi) e gli archi in figura 3.7.b. A causa dei nodi interni ai nuovi archi un circuito hamiltoniano ha solo due modi di attraversare tutti i nodi bianchi, o come illustrato in figura 3.7.c oppure come in figura 3.7.d. Nel primo caso è come se il circuito attraversasse l'arco di assegnamento ma non quello di clausola e nel secondo esattamente l'opposto.

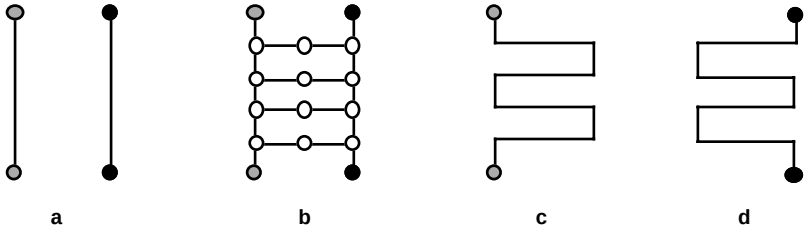


FIGURA 3.7

Quindi bisogna collegare ogni arco del circuito di clausola ad un arco di assegnamento con il dispositivo di figura 3.7. Si noti che a causa del funzionamento del dispositivo l'arco del circuito di clausola viene attraversato se e solo se l'arco di assegnamento non viene attraversato. Quindi un letterale normale viene collegato tramite il dispositivo all'arco di assegnamento che rende vera la variabile corrispondente, mentre un letterale negato viene collegato tramite il dispositivo all'arco di assegnamento che rende falsa la variabile corrispondente.

A questo punto tutti i nodi di clausola e il primo e l'ultimo nodo di assegnamento vengono collegati fra loro in una grande cricca (si vedrà subito il perché). Si veda in figura 3.8 il circuito derivato da (3.6). I nodi neri sono i nodi di clausola, quelli grigi di assegnamento e quelli bianchi di dispositivo. Non sono disegnati tutti gli archi. Bisognerebbe aggiungere gli archi della grande cricca.

Prima di entrare nel dettaglio e vedere che c'è effettivamente corrispondenza biunivoca fra le istanze dei due problemi, si deve verificare che la trasformazione è polinomiale. Se nella formula booleana ci sono n variabili e q letterali, il grafo ha $(n+1)+13\,q$ nodi, cioè un numero lineare nell'istanza della soddisfattibilità. Gli archi sono al più quadratici nei nodi, quindi la trasformazione è polinomiale.

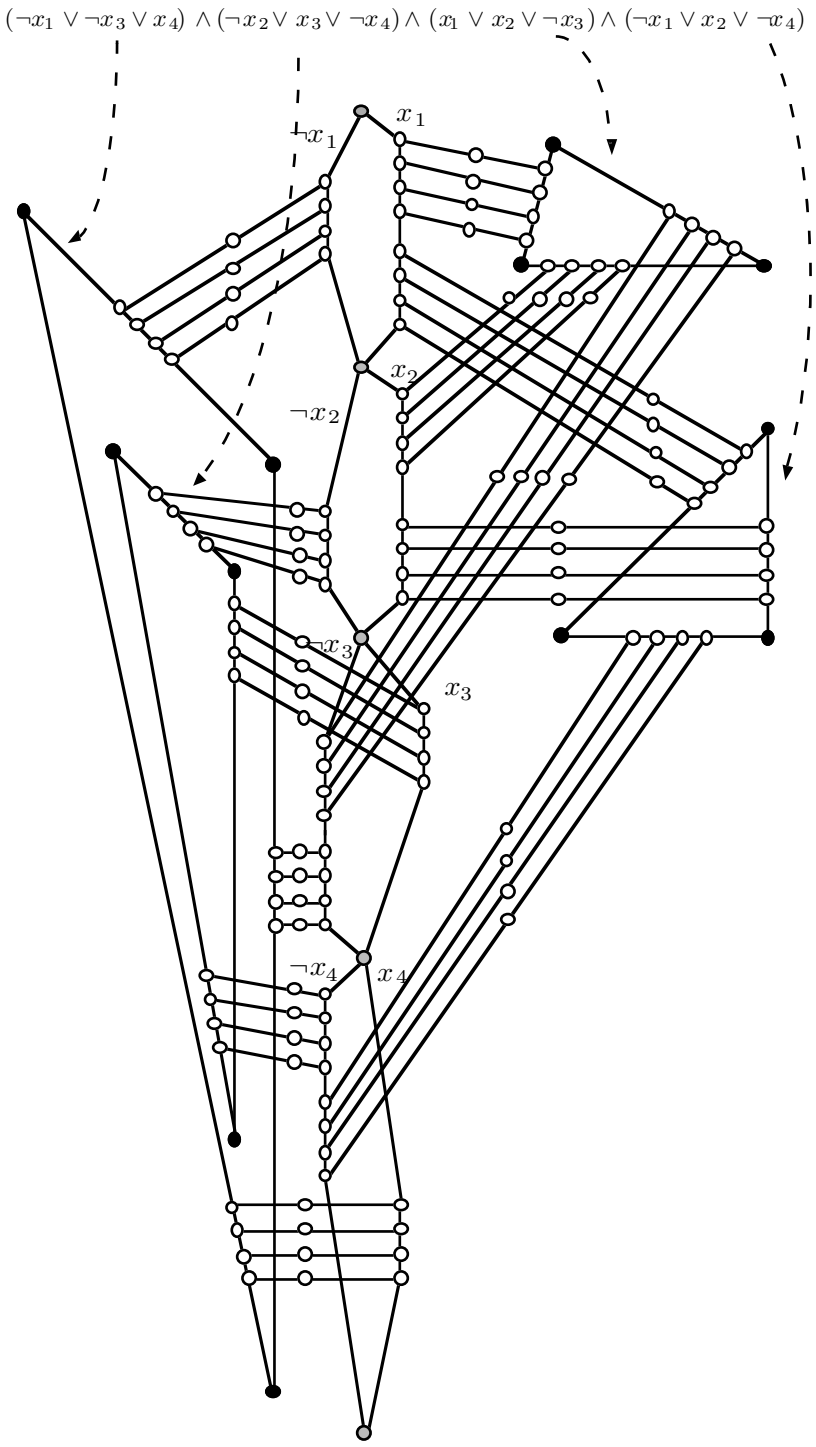

$(\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$
x_1
$\neg x_1$
x_2
$\neg x_2$
$\neg x_3$
x_3
$\neg x_4$
x_4

FIGURA 3.8

Si supponga ora che esista un'assegnazione di valori di verità che renda soddisfatta la formula. Il circuito hamiltoniano procede nel seguente modo: parte dal primo nodo di assegnamento e sceglie l'arco di assegnamento x_1 se x_1 è vera, altrimenti l'arco $\neg x_1$; fatta questa scelta è obbligato a percorrere gli archi delle clausole che si riferiscono alla variabile x_1 (oppure $\neg x_1$). Poi la stessa cosa viene ripetuta con le successive variabili. Finita questa fase rimangono molti nodi non ancora visitati: tutti i nodi di clausola e tutti i nodi dei dispositivi collegati ad archi di assegnamento non attraversati precedentemente. Questi ultimi archi possono essere attraversati solo a partire dai nodi di clausola ed è qui che entra in gioco l'osservazione fatta all'inizio. Se tutti gli archi di un circuito di clausola non sono già stati attraversati (cioè i letterali sono tutti falsi) non c'è modo ammissibile di percorrere il circuito. Però, essendo la formula soddisfatta, abbiamo la garanzia che per ogni circuito di clausola almeno un arco è già stato attraversato e quindi il circuito può attraversare liberamente i rimanenti nodi usando gli archi della grande cricca.

Si supponga ora che esista un circuito hamiltoniano nel grafo. Non è difficile vedere che tale circuito non può attraversare sia un arco di assegnamento associato a x_i che l'altro associato a $\neg x_i$. Inoltre deve attraversarne almeno uno. Si identifichi quest'arco attraversato con il valore di verità da assegnare alla variabile booleana. Per gli stessi ragionamenti visti in precedenza almeno un letterale per clausola deve risultare vero e quindi la formula è soddisfatta. ∎

3.48 Esercizio. Si dimostri che il problema del cammino hamiltoniano è **NP**-completo (si rivedano l'Esempio 1.87 e l'esercizio 1.88). ∎

3.49 Esercizio. Si dimostri che il problema del cammino massimo in un grafo (decidere cioè se esiste un cammino senza ripetizioni di nodi con almeno K archi) è **NP**-completo. ∎

3.50 Esempio. Una volta dimostrato che il circuito hamiltoniano è **NP**-completo è facile dimostrare che il TSP in forma ricognitiva è anch'esso **NP**-completo. Dato un grafo G si costruisca un grafo completo K in cui agli archi presenti in G si assegna costo 0 e agli altri archi costo 1. Ovviamente esiste una soluzione di TSP per K di costo minore o uguale a 0 (di fatto non può essere minore) se e solo se esiste un circuito hamiltoniano in G.

Riguardo al problema se il TSP sia fortemente **NP**-completo si noti che i numeri che compaiono nella trasformazione sono soltanto 0 e 1 e quindi la trasformazione è polinomiale anche rispetto alla codifica unaria. Quindi il TSP è fortemente **NP**-completo. ∎

3.51 Esempio. Ora si consideri il problema della partizione (esempio 1.70): dati n interi positivi si possono partizionare in due insiemi di somma uguale? Per dimostrare che è **NP**-completo bisogna dimostrare, come passo preliminare, che il problema di decidere dell'ammissibilità di

$$\begin{aligned} A\,x &= b \\ x &\in \{0,1\}^n \end{aligned} \tag{3.7}$$

è **NP**-completo (con A matrice $m \times n$). Non è lo stesso problema (3.4) dell'esempio 3.44. Là c'erano diseguaglianze e qui eguaglianze. Naturalmente $A\,x \geq b$ è equivalente a $A\,x - s = b$, $s \geq 0$. In questo modo le diseguaglianze sono trasformate in eguaglianze ma le nuove variabili s_i (una per riga) non sono 0-1. L'idea è allora di decomporre $s_i = y_{i0} + 2\,y_{i1} + 2^2\,y_{i2} + \ldots$, con $y_{ij} \in \{0,1\}$. Ci si deve però preoccupare che la trasformazione sia polinomiale e quindi il numero di variabili y_{ij} che vengono introdotte non può crescere esponenzialmente. Allora si noti che $s_i \leq \sum_j \max\{0, a_{ij}\} - b_i =: B_i$. Quindi bastano $\log_2 B_i$ variabili y_{ij} per ogni i.

Siccome $\log_2 B_i$ è polinomiale in $\sum_j (\lceil \log |a_{ij}| + 1 \rceil + 1) + (\lceil \log |b_i| + 1 \rceil + 1)$ (lunghezza della codifica della riga i), la trasformazione è valida. Ad esempio sia dato

$$\begin{aligned} 3\,x_1 + \;\; x_2 - 2\,x_3 &\geq 0 \\ -\,x_1 + 4\,x_2 + 2\,x_3 &\geq 3 \\ 5\,x_1 - 3\,x_2 + 6\,x_3 &\geq 2 \end{aligned} \tag{3.8}$$

Si ha $s_1 \leq 4$, $s_2 \leq 3$, $s_3 \leq 9$. Allora (3.8) è equivalente a

$$\begin{aligned} 3\,x_1 + \;\; x_2 - 2\,x_3 - x_4 - 2\,x_5 - 4\,x_6 &= 0 \\ -\,x_1 + 4\,x_2 + 2\,x_3 - x_7 - 2\,x_8 &= 3 \\ 5\,x_1 - 3\,x_2 + 6\,x_3 - x_9 - 2\,x_{10} - 4\,x_{11} - 8\,x_{12} &= 2 \end{aligned} \tag{3.9}$$

dove, per motivi di uniformità, si sono indicate con x_k le variabili precedentemente indicate con y_{ij}. Quindi il problema (3.7) è **NP**-completo. Si consideri ora un'aggregazione delle equazioni di (3.7) del seguente tipo:

$$\begin{gathered} \sum_i^m \alpha_i \sum_j^n a_{ij}\,x_j = \sum_i^m \alpha_i\,b_i \\ x \in \{0,1\}^n \end{gathered} \tag{3.10}$$

È chiaro che ogni soluzione di (3.7) è anche soluzione di (3.10) e che normalmente il viceversa non è vero. Tuttavia si consideri, per esempio, l'eguaglianza $100\,a + 2\,b = 200\,c + d$ dove a, b, c, d sono interi non superiori a 10 in valore assoluto. Se $100\,a \neq 200\,c$ i valori di b e c sono troppo piccoli per compensare l'ineguaglianza. Quindi $100\,a = 200\,c$ e quindi anche $2\,b = d$.

L'idea è allora di scegliere i coefficienti α_i in (3.10) in modo che i termini di una riga non possano compensare quelli delle righe successive. Sia

$$B > \max_i \; \max \left\{ \left| \sum_j^n \max\{0, a_{ij}\} - b_i \right|, \left| \sum_j^n \max\{0, -a_{ij}\} - b_i \right| \right\}$$

Allora basta porre $\alpha_i := B^{i-1}$. Di fronte a dei coefficienti che crescono esponenzialmente ci si deve preoccupare che la trasformazione non perda il requisito di essere polinomiale. Se B è scelto in modo 'ragionevole' si ha $\log B = \Theta(\log n + L)$, con L massima lunghezza fra tutte le codifiche dei numeri dell'istanza. Se m è il numero di vincoli si ha $\log \alpha_m = (m-1) \log B$ da cui si vede che la trasformazione è polinomiale (con la codifica normale). Per (3.9) si ha

$$B > \max(\{4, 9\}, \{3, 7\}, \{9, 20\}) = 20$$

e quindi $B := 21$ è sufficiente. Allora (3.9) viene aggregato in

$$\begin{aligned} (3\,x_1 + \;\; x_2 - 2\,x_3 - x_4 - 2\,x_5 - 4\,x_6 &- 0) + \\ 21\,(-x_1 + 4\,x_2 + 2\,x_3 - x_7 - 2\,x_8 &- 3) + \\ 441\,(5\,x_1 - 3\,x_2 + 6\,x_3 - x_9 - 2\,x_{10} - 4\,x_{11} - 8\,x_{12} &- 2) = 0 \end{aligned}$$

cioè

$$\begin{aligned} 2187\,x_1 - 1238\,x_2 + 2686\,x_3 - x_4 - 2\,x_5 - 4\,x_6 - \\ -21\,x_7 - 42\,x_8 - 441\,x_9 - 882\,x_{10} - 1764\,x_{11} - 3528\,x_{12} = 945 \end{aligned}$$

In questa forma il problema non è ancora esattamente la partizione dato che i coefficienti non sono tutti positivi. Ma la cosa non presenta problemi perché basta definire una variabile complementare $\bar{x}_i := (1 - x_i)$ se il coefficiente è negativo. Infine si ha

$$2187\,x_1 + 1238\,\bar{x}_2 + 2686\,x_3 + \bar{x}_4 + 2\,\bar{x}_5 + 4\,\bar{x}_6 + \\ +21\,\bar{x}_7 + 42\,\bar{x}_8 + 441\,\bar{x}_9 + 882\,\bar{x}_{10} + 1764\,\bar{x}_{11} + 3528\,\bar{x}_{12} = 8868$$

Si può obiettare che in questa forma non si tratta ancora della partizione perché il termine noto non è uguale alla metà della somma dei coefficienti. Anche questo problema si risolve facilmente. Supponiamo dapprima che $2\,b < \sum_i^n a_i$. Si può pensare di aggiungere una variabile x_{n+1} con un coefficiente a_{n+1} tale che se questo valore viene aggiunto al termine noto la proprietà sia verificata, ovvero

$$2\,(b + a_{n+1}) = \sum_i^n a_i + a_{n+1}$$

da cui $a_{n+1} = \sum_i^n a_i - 2\,b$. L'istanza con i valori $a_1, \ldots, a_{n+1}$ e termine noto $b' := b + a_{n+1}$ è ora esattamente un problema di partizione. Vogliamo far vedere che

$$\exists x \in \{0,1\}^n : \sum_i^n a_i\,x_i = b \quad \Longleftrightarrow \quad \exists y \in \{0,1\}^{n+1} : \sum_i^{n+1} a_i\,y_i = b'$$

Se esiste x ammissibile basta prendere $y_i := x_i$, $i := 1, \ldots, n$ e $y_{n+1} := 1$ e un tale y è ammissibile. Se viceversa esiste y ammissibile dobbiamo considerare i due casi $y_{n+1} = 0$ e $y_{n+1} = 1$. Se vale l'ultimo caso basta porre $x_i := y_i$, $i := 1, \ldots, n$. Se invece $y_{n+1} = 0$ si ha

$$\sum_i^n a_i\,y_i = b' = b + a_{n+1} = \sum_i^n a_i - b \quad \Longrightarrow \quad \sum_i^n a_i\,(1 - y_i) = b$$

e quindi basta porre $x_i := 1 - y_i$, $i := 1, \ldots, n$. Se ora $2\,b > \sum_i^n a_i$, possiamo considerare l'equivalente problema $\sum_i^n a_i\,(1 - x_i) = b$ e si ragiona di conseguenza.

La cosa interessante, e forse sorprendente, da rilevare è che un problema di programmazione lineare intera è difficile con una come con molte diseguaglianze. Tuttavia, per ridurre le diseguaglianze ad una sola si è stati costretti a costruire coefficienti che crescono esponenzialmente con il numero di variabili e quindi, in qualche modo, non si ottiene una 'tipica' istanza di partizione. Questa considerazione intuitiva ha la sua controparte formale nel fatto che la trasformazione indicata non è polinomiale se si considera la codifica unaria. Si è dimostrata allora la **NP**-completezza, ma non la forte **NP**-completezza del problema della partizione. Infatti si vedrà in seguito che esiste un algoritmo pseudopolinomiale per la partizione. ∎

3.52 Esercizio. Una variante del problema della partizione prevede di ripartire i numeri $a_1, \ldots, a_n$ in due sottoinsiemi non solo di uguale somma ma anche di uguale cardinalità. Dimostrare che anche la variante è **NP**-completa. (suggerimento: nella variante si può sommare ai numeri una quantità costante senza alterare l'istanza...) ∎

Vi sono problemi con numeri nella stringa d'ingresso che sono comunqe fortemente **NP**-completi. Fra questi citiamo il problema della tripartizione (esempio 1.86) (e della variante dell'assegnamento tridimensionale numerico) e del bin packing (vedi esempio 1.76). Si noti

che in versione ricognitiva i problema del bin packing e quello del multiprocessor scheduling sono lo stesso problema.

Le trasformazioni che determinano la **NP**-completezza di questi problemi e di altri che verranno citati sono molto complesse e porterebbero via troppo spazio in questo testo. Il lettore è rinviato alla letteratura citata per le varie dimostrazioni. Oltre ai problemi già citati sono ad esempio **NP**-completi i problemi:

– determinazione del numero cromatico di un grafo: dato un grafo G ed un intero K esiste un modo di colorare il grafo con K o meno colori? (**NP**-completo per $K \geq 3$);

– minimo albero di Steiner (esempio 1.48): dato un grafo G con costi c_e per ogni arco e, un sottoinsieme proprio J di nodi ed un valore K, esiste un albero di supporto su J di costo non superiore a K? Rimane **NP**-completo anche se $c_e = 1$, $\forall e$;

– massimo taglio: nell'esempio 1.50 veniva chiesto di trovare un taglio con il minimo numero di archi. Se si vuole il taglio con il massimo numero di archi il problema (in versione ricognitiva) è **NP**-completo: dato un grafo G ed un intero K, esiste un taglio di G con un numero di archi non inferiore a K?

– assegnamento tridimensionale (esempio 1.83). Rimane **NP**-completo anche nella versione dell'Esercizio 1.85 (triple assegnate tramite coppie).

Sono invece risolubili polinomialmente, come si vedrà in seguito:

– programmazione lineare: data una matrice A e un vettore b, esiste $\hat{x}$ tale che $A\,\hat{x} \leq b$?

– assegnamento (esempi 1.36 e 1.82);

– cammino minimo in un grafo generico con costi non negativi (esempio 1.41);

– minimo albero di supporto (esempio 1.47): dato un grafo G con costi c_e per ogni arco e ed un valore K, esiste un albero di supporto su tutti i nodi di costo non superiore a K?

– minimo taglio (esempio 1.50): dato un grafo G ed un numero K, esiste un taglio di G con un numero di archi non superiore a K?

– accoppiamento pesato (esempio 1.54): dato un grafo completo con costi c_e per ogni arco e ed un valore K, esiste un accoppiamento perfetto di costo non superiore a K?

Vi sono problemi per i quali si può provare la trasformazione da ogni problema in **NP** ma non la loro appartenenza a **NP**. Si tratterebbe ovviamente di problemi **NP**-completi se soltanto si potesse provare che stanno in **NP**. In mancanza di una tale prova si parla di problemi **NP**-*difficili*. Eccone due esempi:

3.53 Esempio. (Papadimitriou et al. [1993]) Sia dato un grafo e due nodi s e t siano identificati come sorgente e destinazione. Ad ogni arco sia associato un numero positivo p_e (tempo di attraversamento dell'arco). Si considerino un certo numero di messaggi: al messaggio i viene associato un cammino da s a t definito dall'elenco degli archi del cammino e ad ogni arco e del cammino viene associato un istante di tempo t_e^i. Un messaggio è ammissibile se $t_e^i + p_e \leq t_{e'}^i$ con e' arco successore di e nel cammino. Un insieme di messaggi ammissibili è ammissibile se $t_e^i + p_e \leq t_e^j$ oppure $t_e^j + p_e \leq t_e^i$ per ogni coppia di messaggi i e j. Fissato un tempo T, esiste un insieme ammissibile di m messaggi?

Una trasformazione è facilmente ottenibile da una variante (sempre **NP**-completa) del problema dell'assegnamento tridimensionale numerico (esempio 1.87) in cui i valori a_i sono vincolati da $B/4 < a_i < B/3$ e altrettanto i valori b_i e c_i. Data un'istanza di quest'ultimo problema si consideri un grafo in cui vi sono quattro nodi s, n_2, n_3 e t. Fra s e n_1 vi sono m archi paralleli con valori p_e uguali a $a_1, \ldots, a_m$. Fra n_1 e n_2 vi sono altri m archi paralleli con valori $b_1, \ldots, b_m$. Similmente fra n_2 e t con valori dati da $c_1, \ldots, c_m$. Basta porre ora $T = B := (\sum_i^m a_i + b_i + c_i)/m$. Se esiste una soluzione al problema dell'assegnamento tridimensionale numerico, questa individua immediatamente m cammini diversi per i messaggi.

Viceversa se esistono m messaggi facciamo vedere che i cammini non possono avere archi in comune. Se così fosse uno dei cammini per essere ammissibile dovrebbe avere un tempo di percorrenza pari alla somma dei tre tempi più il tempo di attesa sull'arco in comune. Siccome i tempi sugli archi sono maggiori di $B/4$ il messaggio non potrebbe arrivare entro il tempo B. Quindi i messaggi usano ciascuno un arco diverso e da questo fatto si deduce immediatamente la soluzione del problema dell'assegnamento.

Tuttavia non si sa se il problema stia in **NP**. Un possibile certificato potrebbe essere rappresentato dall'elenco dei valori t_e^i per ogni messaggio, ma tale elenco contiene almeno m valori e m viene codificato come $\log m$. Quindi il certificato non è succinto. ∎

3.54 Esempio. Questo problema è interessante perché, pur essendo **NP**-difficile e quindi teoricamente non più facile dei problemi **NP**-completi, è pur tuttavia risolvibile da un algoritmo pseudopolinomiale. Dati n interi non negativi $a_1, a_2, \dots a_n$ e due interi non negativi K e B, esistono almeno K sottoinsiemi distinti (non necessariamente disgiunti) $J_1, J_2, \dots, J_K$ di $\{1, 2, \dots, n\}$ tali che $\sum_{i \in J_j} a_i \leq B$ per ogni $j = 1, \dots, K$? Il problema prende il nome di *K insiemi minimi*. Si può trasformare il problema dalla partizione (si noti che la trasformazione non avviene da un problema fortemente **NP**-completo) e quindi è **NP**-difficile. Tuttavia un certificato succinto non è noto. L'elenco dei sottoinsiemi non è ovviamente succinto. ∎

3.55 Esercizio. Si trasformi partizione in K insiemi minimi. ∎

Finora si è fatto sempre riferimento alle versioni ricognitive dei problemi di ottimizzazione in quanto la teoria della **NP**-completezza fa riferimento solo a problemi di decisione. Per poter estendere la teoria anche ai problemi di ricerca (come definiti in 3.10) bisogna introdurre un tipo di trasformazione più ampia che prende il nome di *Turing-riduzione*.

3.56 Definizione. *Un problema A è* Turing-riducibile *ad un problema B se esiste un algoritmo che risolve B ed esiste un algoritmo polinomiale che risolve A usando un numero polinomiale di volte l'algoritmo che risolve B.* ∎

In altre parole l'algoritmo che risolve B è una procedura incorporata all'interno dell'algoritmo che risolve A e il tempo di esecuzione della procedura viene contato come unitario. Quindi se l'algoritmo che risolve B è polinomiale se ne deduce la polinomialità di A. La trasformazione fra problemi di decisione è un particolare caso di Turing-riduzione in cui la procedura viene chiamata una sola volta e c'è corrispondenza fra le istanze 'sì' e 'no'. Possiamo a questo punto estendere il concetto di **NP**-difficoltà anche ai problemi di ricerca.

3.57 Definizione. *Un problema di ricerca si dice* **NP**-difficile *se esiste una Turing-riduzione da un problema di decisione* **NP**-*completo.* ∎

L'esempio 1.78 e l'esercizio 1.79 sono esempi di Turing-riduzioni da problemi di ricerca in problemi di decisione (di ammissibilità). Con procedure simili si possono ricavare Turing-riduzioni dalle versioni di ricerca alle rispettive versioni ricognitive per tutti i problemi **NP**-completi. Molto più semplice è invece una Turing-riduzione dalla versione ricognitiva a quella di ricerca di un problema di ottimizzazione. Infatti trovato l'ottimo è banale poter affermare se esiste o no una soluzione di valore migliore di una costante assegnata. Si dice quindi che il TSP, il minimo albero di Steiner, la minima copertura di nodi ecc. sono problemi **NP**-difficili.

3.8. Altri aspetti di complessità computazionale

In questa sezione vengono trattati in modo informale alcuni aspetti della teoria della complessità computazionale che, pur essendo meno importanti dei precedenti dal mero punto di vista dell'Ottimizzazione, sono però ugualmente interessanti e importanti dal punto di vista informatico. Questi argomenti vengono presentati soprattutto come stimolo per un ulteriore approfondimento nella citata letteratura.

Gli algoritmi fin qui considerati erano sequenziali: un'istruzione alla volta eseguita da un opportuno processore. Come cambia il quadro se si ammettono molti processori che eseguono in parallelo le istruzioni? Vale la pena dedicare più risorse in parallelo alla risoluzione di un determinato problema? Convenzionalmente si è deciso che 'vale la pena' parallelizzare l'esecuzione di un algoritmo se il numero di processori è polinomiale (nella descrizione dell'istanza) e il tempo di esecuzione è logaritmico o almeno polilogaritmico. Si badi che il lavoro totale richiesto, dato dal prodotto fra il numero di processori e il tempo, deve essere polinomiale e pertanto, finché (e probabilmente mai) non si trovino algoritmi polinomiali per i problemi **NP**-completi, il parallelismo non è la 'cura' sperata contro i problemi **NP**-completi.

Il concetto di parallelismo efficiente si applica perciò ai problemi già in **P** e per questi il parallelismo diventa efficiente solo se il tempo di esecuzione viene abbassato drasticamente ad una complessità meno che polinomiale come logaritmica o polilogaritmica. I problemi per i quali si riesca ad ottenere questo risultato formano una nuova classe di complessità detta **NC** ('Nicks's class' in onore di Nicholas Pippenger). La classe **NC** si situa fra **NL** e **P**, cioè $\mathbf{NL} \subset \mathbf{NC} \subset \mathbf{P}$ e, ancora una volta, non si sa se le inclusioni sono strette. Di nuovo i problemi **P**-completi sono quelli per i quali è molto improbabile che il parallelismo riesca ad abbattere i tempi di calcolo a valori polilogaritmici.

3.58 Esempio. La somma di n numeri può essere eseguita efficientemente in parallelo organizzando i calcoli come un torneo ad eliminazione diretta, cioè secondo una struttura ad albero binario. Prima si calcolano $b_1 := a_1 + a_2$, $b_2 := a_3 + a_4$, ..., poi $c_1 := b_1 + b_2$, ..., e così via. Ovviamente il tempo richiesto è logaritmico. Il numero di processori è $n/2$ (si può tuttavia abbassarlo a $O(n/\log n)$. Il principio è ovviamente estendibile a qualsiasi operazione binaria associativa.

Analogamente il prodotto scalare di due vettori richiede tempo logaritmico e, proseguendo, il prodotto di due matrici è anche parallelizzabile, essendo nient'altro che n^3 prodotti scalari di vettori (da eseguire su n^3 processori indipendenti). La potenza k-esima di una matrice è anche parallelizzabile in quanto la potenza k-esima viene calcolata sfruttando ricorsivamente le potenze di 2. ∎

Lo studio della complessità di molti problemi viene spesso affrontato supponendo di introdurre nel modello di computazione degli aspetti stocastici. Consideriamo il seguente esempio:

3.59 Esempio. Sia dato un grafo bipartito $n \times n$, come in figura 3.9 ($n = 5$) e ci chiediamo se esiste un accoppiamento perfetto. A tal fine associamo al grafo bipartito una matrice simbolica in cui l'elemento (i, j) è x_{ij} se esiste nel grafo l'arco (i, j), altrimenti è zero. Quindi al grafo in figura viene associata la matrice

$$A(x) := \begin{pmatrix} 0 & x_{12} & 0 & 0 & x_{15} \\ 0 & x_{22} & 0 & 0 & 0 \\ x_{31} & x_{32} & 0 & x_{34} & 0 \\ x_{41} & 0 & x_{43} & 0 & 0 \\ 0 & x_{52} & 0 & 0 & x_{55} \end{pmatrix}$$

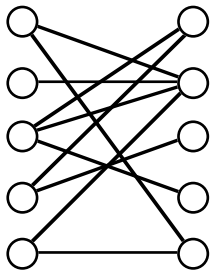


FIGURA 3.9

Il determinante di A è nullo se e solo se il grafo non possiede un accoppiamento perfetto. Infatti il determinante di una matrice può esser scritto come

$$\det A = \sum_{\pi} \sigma(\pi) \prod_{i} a_{i,\pi(i)}$$

dove $\sigma(\pi)$ è il segno della permutazione π e ad ogni permutazione π è associato il monomio $\prod_i a_{i,\pi(i)}$. Per come è costruita A il monomio $\prod_i a_{i,\pi(i)}$ è nullo se almeno uno degli archi $(i, \pi(i))$ è mancante. Allora, se non esiste un accoppiamento perfetto, in ogni monomio deve mancare almeno un arco e quindi tutti i monomi sono nulli e il determinante è nullo. Si noti ora che i monomi, se non nulli, sono tutti diversi e non si possono cancellare nella sommatoria. Quindi il determinante nullo implica tutti i monomi nulli e l'assenza di un accoppiamento perfetto.

Basterebbe allora calcolare il determinante di A e verificare se è nullo. Questo però richiederebbe il calcolo *simbolico* del determinante e questo non si riesce a fare in tempo polinomiale. Tuttavia si può calcolare in modo *numerico* il determinante fissando dei valori per le variabili x_{ij}. Il calcolo numerico si può fare in tempo polinomiale. Si può quindi scegliere a caso un valore x^1. Se avviene $\det A(x^1) \neq 0$ ovviamente esiste un accoppiamento perfetto. Se invece $\det A(x^1) = 0$ due sono i casi possibili: o $\det A(x)$ è identicamente uguale a zero (cioè non esistono accoppiamenti perfetti) oppure x^1 è una radice di $\det A(x)$. Ovviamente non c'è modo di sapere quale dei due casi sia realmente avvenuto anche se si è inclini a pensare poco probabile il secondo caso. Allora si rifaccia l'esperimento con un valore diverso x^2. Di nuovo, se avviene che $\det A(x^2) \neq 0$ c'è un accoppiamento perfetto, altrimenti... è ancora meno probabile che ci sia un accoppiamento perfetto. Si ritenti ancora l'esperimento. Cosa dobbiamo concludere se in k esperimenti diversi $\det A(x^i) = 0$ per $i := 1, \ldots, k$?

Innanzitutto fissiamo una griglia di valori interi da cui vengono presi a caso i vettori x^i. Ad esempio sia $x^i \in \{1, 2, \ldots, K\}^n =: B$ (escludiamo l'origine che è sempre e comunque radice). Al più quanti vettori in B possono essere radici di $\det A(x)$? Siano m le variabili in $\det A(x)$ e sia r_m il numero di radici in B per un polinomio multilineare in m variabili. Se fissiamo arbitrariamente i valori delle variabili $x_1, \ldots, x_{m-1}$ rimane un polinomio di primo grado in x_m se il coefficiente di x_m è non nullo, e quindi in questo caso c'è al più un valore di x_m che annulla il determinante. Questo avviene per ogni possibile scelta in B delle variabili $x_1, \ldots, x_{m-1}$. Quindi al più ci sono K^{m-1} vettori in B che sono radici di $\det A(x)$ se il coefficiente di x_m è non nullo. Se invece il coefficiente di x_m è nullo, e questo può avvenire al più r_{m-1} volte perché il coefficiente è a sua volta un polinomio in $m-1$ variabili, il valore di x_m è irrilevante e quindi vi sono al più $r_{m-1} K$ radici. Allora $r_m \leq K^{m-1} + r_{m-1} K$ con $r_1 \leq 1$. Si ottiene facilmente $r_2 \leq 2K$ e in generale $r_m \leq m K^{m-1}$. La probabilità di sceglierne uno a caso in B è $m K^{m-1}/K^m = m/K$. Quindi basta prendere valori a caso in $\{1, 2, \ldots, 2m\}$ e abbiamo una probabilità minore di $1/2$ di trovare una radice. Dopo k

esperimenti la probabilità di trovare ogni volta una radice scende al di sotto di $1/2^k$ e questo può convincere che un accoppiamento perfetto non esiste. ∎

I problemi che possono essere 'risolti' in tempo polinomiale con uno schema di questo genere costituiscono una nuova classe di problemi, e si può far vedere che tale classe, indicata con **R** (o anche **RP**), dovrebbe stare fra **P** e **NP** (dovrebbe nell'ipotesi $\mathbf{P} \neq \mathbf{NP}$). Un esempio più interessante di quello mostrato (ma anche più complesso) riguarda il problema di decidere se un numero è primo o composto, che, con le conoscenze attuali sta in **NP** ma non in **P** (vedi Papadimitriou [1994] p. 247-253).

Lo schema di iterare una prova fino a raggiungere la convinzione a causa della bassissima probabilità di un evento si può estendere aggiungendo nel modello di computazione un oracolo in grado di eseguire qualsiasi calcolo in tempo unitario.

3.60 Esempio. Se due grafi $G_1 = (N, E_1)$ e $G_2 = (N, E_2)$ sono isomorfi è molto semplice farsi convincere che lo sono. Basta fornire la corrispondenza π fra i nodi e verificare che $(i, j) \in E_1 \iff (\pi(i), \pi(j)) \in E_2$. E questo è quanto dire che il problema sta in **NP**. Supponiamo invece che qualcuno ci voglia convincere che G_1 e G_2 *non* sono isomorfi e noi vogliamo esserne convinti oltre ogni ragionevole dubbio. Il meccanismo che verrà ora introdotto richiede che la persona che ci deve convincere sia una specie di mago (l'oracolo) con potenza di calcolo illimitata. Dobbiamo innanzitutto scegliere uno dei due grafi, in modo casuale e senza rivelare la scelta al mago (che quindi ha potenza di calcolo illimitata ma non è chiaroveggente!). Sempre in modo casuale viene costruito un terzo grafo H, isomorfo a quello scelto e lo si presenta al mago chiedendogli se è isomorfo a G_1 oppure a G_2. Se è vero che G_1 e G_2 non sono isomorfi, H è isomorfo ad uno dei due ma non all'altro e quindi il mago non ha difficoltà a dire quale sia stata la scelta. Però, se G_1 e G_2 sono isomorfi, H è isomorfo ad ambedue e il mago non ha altra scelta che tirare a indovinare. Se dà la risposta sbagliata l'imbroglio del mago è immediatamente smascherato. Se invece dà la risposta giusta due sono i casi: o i grafi sono isomorfi oppure non lo sono e, tirando ad indovinare, il mago ha imbroccato la risposta giusta con probabilità 1/2. Se si ripete l'esperimento k volte e per k volte il mago risponde giusto la probabilità che i grafi siano isomorfi è solo di $1/2^k$.

Questo esempio può anche essere parafrasato scherzosamente pensando ad una persona daltonica che vuole farsi convincere che due disegni sono di colore diverso (e per il resto uguali in tutto). Il daltonico sceglie a caso uno dei due disegni e chiede alla persona non daltonica di quale si tratti, ecc. ∎

L'esempio riguardava il problema dell'isomorfismo fra grafi che non sembra essere **NP**-completo e quindi potrebbe stare in **co-NP**. Tuttavia un meccanismo interattivo simile (anche se molto più complesso) può essere costruito per i problemi **NP**-completi (una descrizione si trova in Johnson [1988]). Non solo, è stato dimostrato che i problemi verificabili in tempo polinomiale con tale meccanismo interattivo coincidono con **PSPACE** (Shamir [1990]). Per una dimostrazione di questo risultato si veda anche Papadimitriou [1994] p. 475.

Vogliamo ora concludere questa panoramica informale con dei risultati che forse costituiscono solo una curiosità anche se molto stimolante.

Si è visto che il certificato che convince della veridicità delle istanze di tipo 'sì' per i problemi **NP**-completi, coincide di fatto quasi sempre con la soluzione medesima. Forse sarebbe più interessante fornire una prova convincente senza però svelare la soluzione. È possibile questo? Effettivamente è possibile, nel senso dei casi precedenti in cui la probabilità di essere ingannati tende a zero. L'idea che sta alla base di tali certificati, detti 'a conoscenza zero', è di mostrare al verificatore solo una piccola parte del certificato tenendo nascosto tutto il resto. Altre parti del certificato verranno mostrate successivamente dopo però aver mescolato ogni volta tutti i dati in modo che il verificatore non riesca ad unire insieme i vari

pezzetti di certificato che gli sono stati mostrati e ricostruire la soluzione.

3.61 Esempio. Si prenda in esame il problema dell'insieme stabile. Chiamiamo R (risolutore) colui che ha risolto il problema e che deve convincere il verificatore, chiamato V, che il grafo possiede almeno K nodi stabili. R crea a caso una copia isomorfa del grafo dicendo apertamente quali sono i nodi stabili nella versione isomorfa. V è autorizzato ad osservare soltanto una coppia di nodi del grafo isomorfo e prende in esame due nodi dichiarati stabili. Se R si è comportato in modo corretto questi nodi non sono adiacenti e V nota che, limitatamente a questa coppia di nodi, i nodi sono stabili. A questo punto R crea a caso ed in modo indipendente dal precedente un'altra copia isomorfa del grafo e la verifica viene ripetuta. Dall'esame di tutte le coppie di nodi scelte da V non c'è modo di inferire quali sono globalmente i nodi stabili. Con che probabilità V può essere stato ingannato? Nel caso peggiore per V vi è un solo arco all'interno dei nodi stabili. Indichiamo $m := K(K-1)/2$. La probabilità di non scegliere questo arco è $1 - 1/m$. Se vengono eseguite $m\,q$ verifiche, la probabilità di non trovare mai l'arco è $(1-1/m)^{mq} \approx e^{-q}$ che può quindi essere resa arbitrariamente piccola con un'opporuna scelta di q. ∎