INTRODUZIONE ALLA COMPLESSITÀ COMPUTAZIONALE

1 – Algoritmi

Dato un ‘problema’ da risolvere, siamo da un lato interessati a trovarne la ‘soluzione’, ma riteniamo ancora più importante trovare dei ‘metodi’ di risoluzione che possano essere ripetutamente applicati al variare dei dati del problema.

Non è certamente pensabile un metodo universale che sia in grado di risolvere qualsiasi problema. 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.

I metodi di risoluzione devono essere di natura algoritmica, cioè devono operare entro un insieme strutturato di regole formalmente definite, in modo da arrivare alla soluzione ‘automaticamente’ e quindi poter essere eseguiti da una macchina. 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 o, più semplicemente, complessità.

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 non sia tanto dovuta 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 un aspetto molto utile della teoria perché permette di fare affermazioni a priori, per certi problemi, 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’. 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. 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 (RAM), 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 Pascal, C, oppure Java.

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 un 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 i 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).

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:

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

$$\begin{aligned} O(f(n)) &:= \Big\{g \in N^N : \textit{esistono } \hat{k},\, c > 0 \textit{ tali che } g(k) \leq c\, f(k), \textit{ per } k \geq \hat{k}\Big\} \\ \Omega(f(n)) &:= \big\{g \in N^N : f \in O(g(n))\big\} \\ \Theta(f(n)) &:= O(f(n)) \cap \Omega(f(n)) \end{aligned}$$

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

$$\hat{\tau}(n) := \max_{s:|s|=n} \tau(s), \qquad \hat{\sigma}(n) := \max_{s:|s|=n} \sigma(s) \tag{1}$$

$$\tilde{\tau}(n) := \min_{s:|s|=n} \tau(s), \qquad \tilde{\sigma}(n) := \min_{s:|s|=n} \sigma(s) \tag{2}$$

$$\bar{\tau}(n) := \sum_{s:|s|=n} p(s)\,\tau(s), \qquad \bar{\sigma}(n) := \sum_{s:|s|=n} p(s)\,\sigma(s) \tag{3}$$

dove $p(s)$ è la probabilità della stringa s.

La definizione (1), detta *misura di caso peggiore*, tiene conto appunto del peggior comportamento dell'algoritmo limitatamente a stringhe di lunghezza fissata. 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 per una opportuna distribuzione di probabilità delle stringhe d'ingresso, come indicato dalle funzioni $\bar{\tau}(n)$ e $\bar{\sigma}(n)$ (*misura di caso medio*). La definizione (2), detta *misura di caso migliore*, tiene conto appunto del miglior comportamento dell'algoritmo limitatamente a stringhe di lunghezza fissata. Questa misura, anche se meno importante di (1), serve per capire quali differenze un algoritmo possa esibire di fronte a diverse stringhe d'ingresso (se ad esempio $\tilde{\tau}(n) = \hat{\tau}(n)$ ovviamente si ha $\tilde{\tau}(n) = \bar{\tau}(n) = \hat{\tau}(n)$).

La misura di gran lunga più importante, e quindi quella a cui si fa normalmente riferimento, è quella di caso peggiore, essenzialmente per le seguenti ragioni: limitazioni di caso peggiore sono molto più facili da ottenere di quelle di caso medio ed è molto difficile giustificare teoricamente l'uso di una particolare distribuzione di probabilità sulle stringhe d'ingresso.

Una valutazione diretta delle funzioni in (1), (2) e (3) 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 $\hat{\sigma}(n) \in O(f)$), $\tilde{\tau} \in \Omega(f)$ (oppure $\tilde{\sigma} \in \Omega(f)$), $\bar{\tau} \in \Theta(f)$ (oppure $\bar{\sigma} \in \Theta(f)$).

Gli algoritmi vengono divisi in due grandi classi secondo la seguente definizione che fa riferimento alla complessità temporale:

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. ∎

Meno importante è la medesima distinzione rispetto alla complessità spaziale:

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 $\hat{\tau}(n) \leq C \in O(1)$ 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* esiste n tale che $\hat{\tau}(n) = \infty$.

4 – ESERCIZIO. Dimostrare che le classi elencate sono incluse strettamente una nell'altra. ∎

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 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. 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.

6 – Esempio. Siano dati m numeri interi $a_1, a_2, \ldots, a_m$. Supponiamo di sapere che i numeri sono tutti compresi nell'intervallo $[1, K]$ con $K = 2^p - 1$ e quindi ogni numero viene codificato con $\log_2 K$ bit. La stringa d'ingresso è quindi formata da $m \log_2 K$ simboli. Si vuole trovare il più grande numero, cioè la stringa d'uscita deve contenere il numero $\max_i a_i$ ($\log_2 K$ simboli).

Un possibile algoritmo, detto di *scansione lineare*, legge i numeri ad uno ad uno confrontandoli con il migliore fra tutti quelli già letti. Le operazioni da eseguire sono pertanto quelle di leggere il numero e di confrontarlo con un altro numero.

```
Algoritmo di ricerca massimo con scansione lineare
main
    input(a_1, a_2, ..., a_m);
    max=0;
    for i := 1 to m do
        if a_i >max then max= a_i;
    output(max)
end.
```

Sia $n = m \log_2 K$ la lunghezza della stringa d'ingresso. L'algoritmo deve leggere tutti i simboli (n operazioni) e deve eseguire $m = n/\log_2 K$ confronti. Quindi

$$\hat{\tau}(n) = \tilde{\tau}(n) = \bar{\tau}(n) = n + \frac{n}{\log_2 K} \in \Theta(n)$$

La risorsa di spazio (solo memoria intermedia!) è data solo dai simboli necessari a codificare max, quindi $\log_2 K$, un valore costante ed indipendente dalla lunghezza della stringa d'ingresso. Quindi

$$\hat{\sigma}(n) = \tilde{\sigma}(n) = \bar{\sigma}(n) = \log_2 K \in \Theta(1)$$

Questo esempio costituisce un caso abbastanza raro in cui la memoria interna è costante al variare dei dati del problema. Normalmente ciò non avviene. ∎

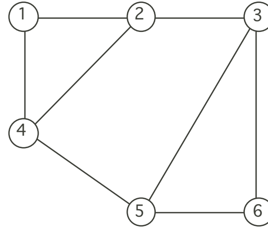


Figura 1

7 – Esempio. Sia assegnato un grafo $G = (N, E)$ (un esempio in Figura 1). Vogliamo determinare se è connesso. La stringa d'ingresso che codifica il grafo può essere semplicemente l'elenco degli archi, definiti come coppie non ordinate di nodi. I nodi vengono identificati da un numero. Quindi se il grafo ha n nodi e m archi, ogni nodo viene identificato da una stringa di $\log n$ simboli e quindi tutta la stringa d'ingresso ha lunghezza $L := 2\,m\,\log n \geq m \geq n-1$ (se $m < n-1$ il grafo è necessariamente sconnesso). Per valutare la complessità secondo i parametri n o m, sia $m = f(n)$. Se $f(n) \in \Theta(n)$ (grafo sparso) allora $L \in \Theta(n\,\log n)$ e $L \in \Theta(m\,\log m)$ da cui $n \in O(L) \cap \Omega(\sqrt{L})$ e altrettanto per m. Se invece $f(n) \in \Theta(n^2)$ (grafo denso) allora $L \in \Theta(n^2\,\log n)$ e $L \in \Theta(m\,\log \sqrt{m}) = \Theta(m\,\log m)$, da cui $n \in O(\sqrt{L}) \cap \Omega(\sqrt[3]{L})$ e $m \in O(L) \cap \Omega(\sqrt{L})$. Per l'esempio in figura la stringa d'ingresso potrebbe essere

$$1, 2, 2, 3, 1, 4, 4, 2, 4, 5, 5, 3, 3, 6, 5, 6$$

L'algoritmo esegue una prima lettura dei dati d'ingresso creando per ogni nodo la lista d'adiacenza, cioè l'elenco dei nodi adiacenti, quindi

$$\begin{array}{ccc} 1 & \rightarrow & 2, 4 \\ 2 & \rightarrow & 1, 3, 4 \\ 3 & \rightarrow & 2, 5, 6 \\ 4 & \rightarrow & 1, 2, 5 \\ 5 & \rightarrow & 4, 3, 6 \\ 6 & \rightarrow & 3, 5 \end{array}$$

Poi i nodi vengono suddivisi in tre insiemi Q, M e P, dove Q identifica i nodi non ancora marcati dall'algoritmo, M i nodi marcati ma non ancora processati e P i nodi processati. Inizialmente $Q := N \setminus \{1\}$, $M := \{1\}$ e $P := \emptyset$. Poi l'algoritmo sceglie un nodo dall'insieme M (sia k tale nodo), lo sposta in P e sposta in M tutti i nodi in Q adiacenti a k. La procedura termina quando M oppure Q sono vuoti. Se Q è vuoto il grafo è connesso, altrimenti i nodi in Q rappresentano i nodi del grafo non connessi con il nodo 1. La correttezza dell'algoritmo è lasciata come facile esercizio.

Per ciò che riguarda l'implementazione dell'algoritmo, l'appartenenza ad uno dei tre sottoinsiemi può essere definata da un vettore S (stato) in cui in posizione i-ma si trova uno dei tre simboli Q, M e P. Inoltre è utile poter disporre esplicitamente dell'insieme M come elenco di nodi dal quale poter prelevare il nodo con complessità $O(1)$ e di un contatore p che indica la cardinalità di Q. Gli insiemi Q e P possono non essere rappresentati esplicitamente. Inizialmente

$$S := (M, Q, Q, Q, Q, Q), \qquad M := \{1\}\,, \qquad p := 5$$

L'algoritmo procede allora nel seguente modo: sceglie (necessariamente) $k := 1$, togliendolo subito da M (che diventa vuoto) e cambiando in P il suo stato, poi scandisce ad uno ad uno gli elementi della lista d'adiacenza di k (in questo caso 2 e 4) e si esamina lo stato di ognuno. Se lo stato è Q il nodo viene aggiunto ad M, il suo stato cambiato in M e p viene diminuito di uno, altrimenti si passa al nodo successivo. Quindi la situazione dopo questa prima iterazione è

$$S := (P, M, Q, M, Q, Q), \qquad M := \{2, 4\}\,, \qquad p := 3$$

Nella successiva iterazione si sceglie il nodo $k = 4$. Della lista d'adiacenza solo il nodo 5 viene aggiunto ad M e quindi si ha

$$S := (P, M, Q, P, M, Q), \qquad M := \{2, 5\}, \qquad p := 2$$

Poi $k = 5$ da cui anche i nodi 3 e 6 vanno in M. A questo punto $p = 0$ e quindi l'algoritmo termina, dichiarando il grafo connesso.

Come si vede il numero di operazioni dell'algoritmo è in ogni caso limitato superiormente dal doppio del numero di archi. Infatti gli elementi della lista d'adiacenza vengono esaminati al più una volta. Quindi sia per grafi sparsi che per grafi densi

$$\hat{\tau}(L) \in O(L)$$

Inoltre il numero di operazioni è limitato inferiormente dal numero di nodi (non si può determinare la connessione senza aver visitato tutti i nodi). Il caso migliore è quindi rappresentato da un grafo completo in cui si determina la connessione con solo n operazioni e quindi

$$\tilde{\tau}(L) \in \Omega(\sqrt[3]{L})$$

La complessità computazionale va, correttamente, misurata rispetto alla lunghezza della stringa d'ingresso. Tuttavia, può essere spesso più semplice e conveniente misurarla rispetto ad un altro parametro del problema strettamente correlato (in modo polinomiale) con la lunghezza della stringa, come in questo esempio sono n oppure m. Finché questa semplificazione nella valutazione della complessità non altera la natura polinomiale (o non polinomiale) di un algoritmo è sempre lecito operare in questo modo. Quindi possiamo più semplicemente dire che l'algoritmo per determinare la connessione di un grafo ha complessità $O(m)$. ∎

8 – 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 si trova un MCD che risulta anche essere 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

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

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 :

$$d_{k+1} \geq d_k + d_{k-1},\ k = 1, \ldots, n \tag{4}$$

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 (4) 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 $n \in O(\log b)$. Il numero di divisioni da eseguire è pertanto lineare in $\log b$ e siccome la lunghezza della stringa d'ingresso (cioè il numero di simboli che definiscono a e b) è $\log a + \log_b$, il numero di divisioni è lineare. Si può dimostrare che eseguire la divisione è un'operazione polinomiale (rispetto alla lunghezza dei numeri da dividere) e pertanto l'algoritmo di Euclide è polinomiale. ∎

9 – Esempio. (Problema della Torre di Hanoi) Ci sono tre pioli e n dischi infilati sui pioli. Una configurazione di dischi viene detta ammissibile se e solo se ogni disco poggia su un disco più grande. Si supponga che tutti i dischi siano infilati su un piolo. Bisogna spostare tutti i dischi su un altro piolo muovendoli uno alla volta e generando soltanto configurazioni ammissibili. Vogliamo generare l'algoritmo che esegue questo compito (ammesso che sia possibile eseguirlo) e valutarne la complessità.

Dimostriamo in modo induttivo che è possibile eseguire il compito. Dapprima dimostriamo che è possibile spostare n dischi se è possibile spostarne $n-1$. Infatti, per spostare il disco più grande bisogna preventivamente spostare tutti gli $n-1$ dischi sull'altro piolo, poi il disco stesso e infine nuovamente la pila di $n-1$ dischi sul disco appena spostato. Siccome è banalmente possibile spostare due dischi, per induzione si ottiene che è possibile farlo per ogni n. La dimostrazione ricorsiva porta anche al seguente algoritmo ricorsivo:

```
Algoritmo della Torre di Hanoi
main
    input(n)
    move(1,3,n)
end;
procedure move(i, j, n)
begin
    if n = 1
    then put top element of i to j
    else begin
        let k ≠ i and k ≠ j;
        move(i, k, n − 1);
        move(i, j, 1);
        move(k, j, n − 1);
    end
end
```

In Figura 2 è raffigurata la computazione per $n = 4$ con tutti i dischi inizialmente sul primo piolo da spostarsi tutti sul terzo. Ogni nodo (o, tranne il nodo radice, equivalentemente ogni arco) rappresenta una chiamata della procedura 'move'. I tre valori accanto agli archi sono i tre parametri con cui viene chiamata la procedura. Lo spostamento di un disco avviene quando il terzo parametro della procedura 'move' vale 1 ed è raffigurato dai dischi neri.

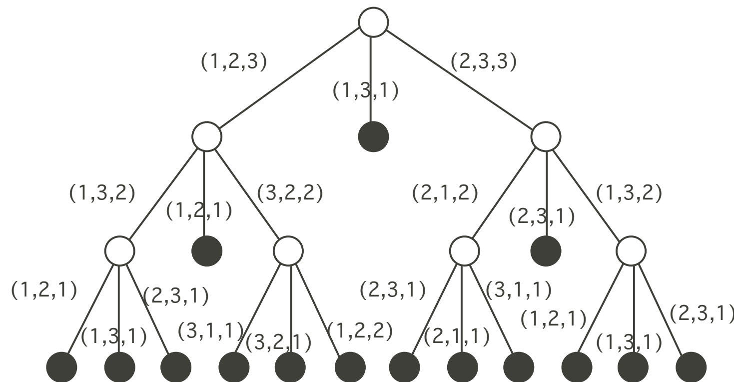


Figura 2

Sia allora $h(n)$ il numero minimo di mosse per spostare una torre di n dischi. Allora in base al ragionamento precedente si ha che $h(n) \leq h(n-1) + 1 + h(n-1) = 2\,h(n-1) + 1$. Inoltre, per ogni successione di mosse, inclusa la minima, si deve passare attraverso la configurazione in cui il disco più grande si può spostare

e quindi tutti gli altri $n-1$ dischi sono sull'altro piolo. Questa successione di mosse ne richiede $2\,h(n-1)+1$ e questo valore è una limitazione inferiore per ogni successione di mosse, quindi $h(n) \geq 2\,h(n-1)+1$ da cui

$$h(n) = 2\,h(n-1) + 1 \tag{5}$$

Inoltre $h(0) = 0$. La ricorsione (5) fornisce $h(1) = 1$, $h(2) = 3$, $h(3) = 7$, $h(4) = 15$, $h(5) = 31$, $h(6) = 63$, ecc. Sorge il sospetto che $h(n) = 2^n - 1$. Per dimostrarlo si definisca $h'(n) = h(n) + 1$. Allora (5) diventa

$$h'(n) - 1 = 2\,(h'(n) - 1) + 1 \qquad \Longrightarrow \qquad h'(n) = 2\,h'(n)$$

con $h'(0) = 1$. Ovviamente $h'(n) = 2^n$ da cui $h(n) = 2^n - 1 \in \Theta(2^n)$.

Per valutare la complessità dell'algoritmo consideriamo come operazione elementare una chiamata della procedura (ovvero il numero di nodi dell'albero di Figura 2).

Sia $\tau(n)$ il numero di chiamate della procedura. In base alla ricorsione si ha

$$\tau(n) = 2\,\tau(n-1) + 2 \tag{6}$$

cioè la chiamata della procedura con parametro n comporta oltre alla chiamata stessa, due chiamate della procedura con parametro $(n-1)$ (con tutte le chiamate che ne deriveranno) ed una chiamata della procedura con $n = 1$ (spostamento di un disco). La condizione iniziale è data da $\tau(1) = 1$. Riscriviamo (6) come

$$\tau(n) + 2 = 2\,\tau(n-1) + 4 = 2\,(\tau(n-1) + 2)$$

Poniamo $\tau'(n) = \tau(n) + 2$ con $\tau'(1) = 3$. Allora $\tau'(n) = 2\,\tau'(n-1)$ per $n \geq 2$ e quindi $\tau'(n) = 3 \cdot 2^{n-1}$ da cui

$$\tau(n) = 3 \cdot 2^{n-1} - 2 = (1+2)\,2^{n-1} - 2 = 2^n - 1 + 2^{n-1} - 1 = h(n) + h(n-1) \in \Theta(2^n)$$

Si noti che si è valutata la complessità usando n come misura della stringa d'ingresso. In realtà la stringa d'ingresso può essere ridotta alla codifica del numero n (e non già alla sequenza $\{1, 2 \ldots, n\}$ che rappresenterebbe i pioli) e ciò richiede solamente $m := \log_2 n$ simboli. Ragionando in questo modo la complessità diventa ancora più elevata, ovvero

$$\tau(m) = 2^{2^m} - 1 \in \Theta(2^{2^m})$$

Per ciò che riguarda la complessità spaziale si noti che ad ogni chiamata della procedura 'move' vanno memorizzati i valori i e j utilizzati dalla procedura chiamante. Con riferimento alla Figura 2 bisogna memorizzare i parametri del cammino dal nodo corrente alla radice. Quindi la risorsa di spazio è $\sigma(n) \in \Theta(n)$.

L'algoritmo della Torre di Hanoi, per quanto visto, non può essere accelerato dato che il numero di spostamenti di dischi è il numero $h(n)$ che esibisce una crescita esponenziale. Per comprendere quanto sia intrattabile un algoritmo con crescita esponenziale, si immagini che lo spostamento di un disco possa avvenire in un nanosecondo (10^{-9} sec corrispondenti ad un clock da 1 GHz). Allora il tempo richiesto per spostare le torri avverrebbe con i seguenti tempi per vari valori n:

$n = 16$	0.000065535 sec
$n = 32$	4.29497 sec
$n = 48$	3 giorni 6 ore 11 min 15 sec
$n = 64$	584 anni 343 giorni 23 ore 34 min 34 sec
$n = 80$	38334786 anni 96 giorni 6 ore 43 min 49 sec

L'inventore del problema della Torre di Hanoi è il matematico francese Edouard Lucas che nel 1883 costruì fisicamente il gioco con 8 dischi suggerendo anche una leggenda secondo la quale dei monaci avrebbero il compito di spostare non 8 ma 64 dischi implicando la fine del mondo alla fine del lavoro. Con tempi 'fisici' di trasposto di un disco (anziché il nanosecondo del calcolatore) il tempo richiesto supererebbe di gran lunga l'età presunta dell'universo. ∎

10 – Esercizio. Si dimostri che la relazione ricorsiva

$$\tau(n) = a\,\tau(n-1) + b$$

dà luogo all'espressione in forma chiusa

$$\tau(n) = a^n\,\tau(0) + \frac{(a^n - 1)\,b}{a-1}$$

per $a \neq 1$. Quale è l'espressione di $\tau(n)$ per $a = 1$? ∎

11 – Esempio. I numeri di Fibonacci sono definiti dalla seguente relazione ricorsiva:

$$F_n = F_{n-1} + F_{n-2}, \qquad F_0 = 0, \qquad F_1 = 1$$

La relazione stessa suggerisce il seguente algoritmo ricorsivo:

```
Algoritmo ricorsivo per i numeri di Fibonacci
main
        input(n);
        output(fibonacci(n))
end;
procedure fibonacci(n)
begin
        if n = 0
        then return(0)
        else if n = 1
                then return(1)
                else return(fibonacci(n − 1)+fibonacci(n − 2))
end
```

Il calcolo di $\tau(n)$ può essere effettuato anch'esso ricorsivamente. Infatti $\tau(0) = 1$ (il test $n = 0$) e $\tau(1) = 2$ (i due test $n = 0$ e $n = 1$) mentre

$$\tau(n) = 3 + \tau(n-1) + \tau(n-2)$$

(i due test più la somma e i tempi relativi alle due chiamate). Ponendo $\tau'(n) = \tau(n) + 3$ si ha

$$\tau'(n) = \tau'(n-1) + \tau'(n-2), \qquad \tau'(0) = 4, \qquad \tau'(1) = 5$$

$$\tau'(0) = F_0 + 4 = F_0 + 4\,F_1, \qquad \tau'(1) = F_1 + 4 = F_1 + 4\,F_2, \qquad \tau'(2) = F_0 + F_1 + 4\,(F_1 + F_2) = F_2 + 4\,F_3,$$

$$\tau'(3) = F_2 + F_1 + 4\,(F_3 + F_2) = F_3 + 4\,F_4 \quad\Longrightarrow\quad \tau'(n) = F_n + 4\,F_{n+1} \quad\Longrightarrow\quad \tau(n) = F_n + 4\,F_{n+1} - 3$$

Si possono ottenere delle limitazioni superiori ed inferiori ai numeri di Fibonacci nel seguente modo

$$F_n = F_{n-1} + F_{n-2} \quad\Longrightarrow\quad F_n \geq 2\,F_{n-2}$$

Se n è dispari, da $F_1 = 1$ si ha $F_3 \geq 2$, $F_5 \geq 2\,F_3 \geq 2^2$, $F_n \geq 2^{(n-1)/2}$. Se n pari si ha, da $F_2 = 1$, $F_4 \geq 2$, $F_6 \geq 2\,F_4 \geq 2^2$, $F_n \geq 2^{(n-2)/2}$. Siccome $2^{(n-2)/2} < 2^{(n-1)/2}$, si ha, per ogni n, $F_n \geq 2^{(n-2)/2}$. Analogamente si ottiene

$$F_n = F_{n-1} + F_{n-2} \quad\Longrightarrow\quad F_n \leq 2\,F_{n-1}$$

Da $F_2 = 1$ si ha $F_3 \leq 2$, $F_4 \leq 2\,F_3 = 2^2$, $F_n \leq 2^{n-2}$. Quindi

$$2^{(n-2)/2} \leq F_n \leq 2^{n-2} \qquad\Longrightarrow\qquad F_n \in O(2^n), \quad F_n \in \Omega(\sqrt{2^n})$$

Aggiungiamo che un calcolo alquanto complesso, basato sulle funzioni generatrici, porta alla seguente espressione esatta per i numeri di Fibonacci

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n \in \Theta\Big((\frac{1+\sqrt{5}}{2})^n \Big) \tag{7}$$

Allora

$$\tau(n) = F_n + 4\,F_{n+1} - 3 \geq 2^{(n-2)/2} + 4\,2^{(n-1)/2} - 3 = 2^{n/2}\,2^{-1} + 2^2\,2^{n/2}\,2^{-1/2} - 3 \in \Omega(\sqrt{2^n})$$

Si tratta di una complessità esponenziale! Siamo proprio costretti ad eseguire i calcoli in modo così dispendioso? Dove si trova il difetto dell'algoritmo?

A differenza del problema della Torre di Hanoi dove non c'è possibilità di abbassare il numero di iterazioni, il calcolo del n-mo numero di Fibonacci può essere accelerato notevolmente. Si può subito notare come la relazione ricorsiva sia estrememante dispendiosa perché ripete il calcolo di un numero di Fibonacci molte volte, mentre, una volta calcolato, lo si può memorizzare e poi usare quando serve. Un meccanismo iterativo di calcolo è quindi molto più efficiente.

```
Algoritmo iterativo per i numeri di Fibonacci
main
    input(n);
    if n = 0
    then output(0)
    else if n = 1
        then output(1)
        else
        begin
            a := 0; b = 1;
            for k := 2 to n do
            begin
                c := a + b; b := c; a := b
            end;
        end
    output(c)
end
```

È chiaro che la complessità dell'algoritmo è $O(n)$. Si noti che tale complessità *non* è lineare perché n si può semplicemente codificare con $\log n$ simboli. Si può allora migliorare la complessità? L'espressione (7) sembra suggerire un metodo più semplice di calcolo. Senza usare (7) che richiede di operare con dei numeri irrazionali introducendo così dei problemi di approssimazione e rappresentazione che non vogliamo qui affrontare, osserviamo che i numeri F_n, F_{n-1} possono essere ottenuti dai numeri F_{n-1}, F_{n-2} da

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Il problema si trasforma pertanto nel calcolo della potenza n-ma di una matrice 2×2. Questa viene eseguita efficacemente calcolando iterativamente:

$$A_0 := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \qquad A_k := A_{k-1}^2 \qquad k := 1, \ldots, p, \qquad p := \lfloor \log_2(n-1) \rfloor$$

Dopodiché, se $b_0, b_1, \ldots, b_p$ sono le cifre binarie di $n-1$ (cioè $n-1 = \sum_{k=0}^{p} b_k\, 2^k$) basta eeguire il prodotto

$$\prod_{k: b_k = 1} A_k$$

Ad esempio per $n = 50$ si calcola $p = 5 = \lfloor \log_2 49 \rfloor$ e poi:

$$A_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \qquad A_2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}^2 = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}, \quad A_3 = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}^2 = \begin{pmatrix} 34 & 21 \\ 21 & 13 \end{pmatrix}$$

$$A_4 = \begin{pmatrix} 34 & 21 \\ 21 & 13 \end{pmatrix}^2 = \begin{pmatrix} 1597 & 987 \\ 987 & 610 \end{pmatrix}, \qquad A_5 = \begin{pmatrix} 1597 & 987 \\ 987 & 610 \end{pmatrix}^2 = \begin{pmatrix} 3524578 & 2178309 \\ 2178309 & 1346269 \end{pmatrix}$$

Le cifre binarie di 49 sono: $b_0 = 1$, $b_1 = 0$, $b_2 = 0$, $b_3 = 0$, $b_4 = 1$, $b_5 = 1$, quindi si esegue

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1597 & 987 \\ 987 & 610 \end{pmatrix} \begin{pmatrix} 3524578 & 2178309 \\ 2178309 & 1346269 \end{pmatrix} = \begin{pmatrix} 12586269025 & 7778742049 \\ 7778742049 & 4807526976 \end{pmatrix}$$

da cui $F_{50} = 12586269025$.

Il prodotto di due matrici 2×2 richiede 8 moltiplicazioni e 4 somme, quindi 12 operazioni aritmetiche. Allora per calcolare le matrici $A_1, \ldots, A_p$ sono richieste $12 \lfloor \log_2 n \rfloor$ operazioni aritmetiche. Il successivo calcolo può richiedere al massimo il prodotto di $p + 1$ matrici, che costa $12\,p = 12 \lfloor \log_2 n \rfloor$ operazioni aritmetiche, da cui il numero di operazioni aritmetiche è $O(\log n)$, cioè lineare rispetto alla codifica della stringa d'ingresso.

Tuttavia bisogna anche valutare la complessità delle operazioni aritmetiche. Si può osservare che i numeri delle matrici crescono molto rapidamente (i valori della matrice A_k sono proprio i numeri di Fibonacci F_m, F_{m+1} e F_{m-1} con $m = 2^k$) e quindi dobbiamo tenere espressamente conto che le operazioni aritmetiche diventano più lente all'aumentare delle cifre dei numeri. Se due numeri hanno p e q cifre il loro prodotto richiede $p \cdot q$ prodotti di singole cifre e altrettante somme. Dalla formula (7) abbiamo

$$\log_2 F_m \approx \log_2 \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^m = \log_2 \frac{1}{\sqrt{5}} + m \log_2 \left(\frac{1 + \sqrt{5}}{2} \right) \in O(m)$$

Siccome $\log_2 F_m$ è il numero di cifre di F_m, i valori della matrice A_k hanno $O(2^k)$ cifre e quindi il calcolo della matrice A_{k+1} richiede $O(2^{2k})$ operazioni sulle cifre. Allora solo il calcolo di A_p, con $p = \lfloor \log_2 n \rfloor$, richiede $O(2^{2(p-1)}) = O(2^{2 \log_2 n/2)}) = O(2^{\log_2 (n/2)^2)}) = O((n/2)^2) = O(n^2)$ operazioni sulle cifre. Questo valore di complessità non è polinomiale perché la stringa d'ingresso ha solo $\log_2 n$ simboli. ∎

3 – 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. 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 dell'istanza.

12 – 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 astratta definizione 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'. 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 soluzioni per la stessa istanza i, u e u' potrebbero essere le due soluzioni).

Un problema, come definito in 12, 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 standardizzare la risposta dell'istanza. Si passa quindi a considerare. una classe particolare di problemi che riveste un ruolo fondamentale nella teoria della complessità.

13 – 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.

14 – 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.

15 – 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 le 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, mescolando 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? ▮

16 – Esempio. Dato un problema di ottimizzazione

$$\begin{aligned} \min \quad & f(x) \\ & x \in F \end{aligned}$$

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. ▮

17 – 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. Una delle classi più importanti è la seguente:

18 – Definizione. *La classe* **P** *è l'insieme dei problemi di decisione risolubili da un algoritmo polinomiale nel tempo.* ▮

I problemi in **P** 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 nella maggior parte dei casi la cosa non riveste un aspetto essenziale. Normalmente se esiste un algoritmo polinomiale per un problema di decisione ne esiste anche uno per la versione di ricerca (nell'altro senso la cosa è banalmente vera). Tuttavia va rilevato come sia stato dimostrato (estate 2002) che il problema di decidere se un numero è primo o composto sta in **P**, ma non si sa ancora se trovare i fattori di un numero composto possa essere eseguito in tempo polinomiale. Si tratta di un caso forse unico di problema di decisione la cui controparte di ricerca non è altrettanto nota e si tratta come ben noto di un problema di grande rilevanza pratica dato che tutti gli algoritmi di crittografia moderna si basano proprio sulla difficoltà di fattorizzare numeri molto grandi.

4 – Algoritmi non deterministici e classe NP

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.

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: 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 a questa verifica, 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 viene chiamato *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*.

19 – 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.

20 – 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**.

21 – 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. ∎

22 – Esempio. Si consideri un sistema di disequazioni lineri $A\,x \le b$. Ci si chiede se esiste una soluzione ammissibile intera (Problema della Programmazione Lineare Intera in versione ricognitiva). 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 sia dato il seguente sistema:

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

con a e b interi positivi, dove 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, come si sarà notato. 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 a **NP**.

23 – 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**.

24 – 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**.

25 – Esempio. Dato un intero, è primo? Se non è primo, basta fornire una fattorizzazione. Quindi il problema della primalità sta in **co-NP**. ∎

26 – 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. ∎

27 – Esempio. (vedi Esempio 16) 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'. Quindi si ha

$$\mathbf{P} \subseteq \mathbf{NP} \tag{8}$$

Le domande fondamentali riguardano la validità delle seguenti eguaglianze

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

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

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

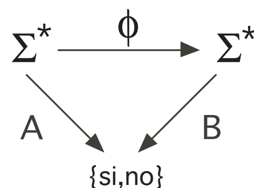
(che invece non implicherebbe $\mathbf{P} = \mathbf{NP}$). Le due questioni (insieme a diverse altre analoghe) sono a tutt'oggi irrisolte. Tuttavia si pensa, sulla base dei risultati parziali ottenuti in tutti questi anni, che l'inclusione (8) sia stretta e che $\mathbf{NP} \neq \mathbf{co\text{-}NP}$.

Di tutte le 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, cercando di trovare un algoritmo polinomiale per almeno un problema **NP**-completo (vedi più avanti), tanto che vi sono stati molti tentativi di ricerca, tutti falliti. Considerato l'enorme sforzo investito in questi trenta 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 $\mathbf{P} = \mathbf{NP}$?

5 – Trasformazioni e completezza

I risultati precedenti fanno vedere come esistano problemi facili (quelli in **P**), e problemi in **NP** ma probabilmente non in **P** e quindi difficili. Finché non sarà risolta la congettura che **P** è contenuta strettamente in **NP** non si potrà affermare che vi sono problemi in **NP** ma non in **P**. Tuttavia si può definire una classe di problemi che molto probabilmente non sta in **P**, perché, se solo uno di questi problemi stesse in **P** allora si avrebbe $\mathbf{P} = \mathbf{NP}$

Per questa analisi serve 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. L'algoritmo di trasformazione deve essere 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 $A \leq_P C$.

Si supponga ora di sapere che tutti i problemi di **P** sono trasformabili tramite $\leq_P$ in un determinato problema A. Se si ottenesse per A un algoritmo polinomiale l'effetto di questo risultato sarebbe notevole perché automaticamente la classe **NP** collasserebbe in **P** a causa della transitività della trasformazione. A causa di questo effetto è lecito riguardare problemi A con queste caratteristiche i più rappresentativi, ovvero i più difficili, della classe **NP**, in quanto una loro risoluzione più efficiente si propaga su tutti gli altri problemi della classe. Abbiamo allora la seguente definizione:

28 – 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**. ∎

I problemi **NP**-completi sono verosimilmente non polinomiali, altrimenti **NP** collasserebbe in **P**, e quindi si tratta di problemi la cui difficoltà è 'garantita'. Per questi motivo 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:

29 – TEOREMA. (S. Cook, 1971) *Il problema della Soddisfattibilità è* **NP***-completo.* ∎

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 alcuni esempi nella prossima sezione. Una definizione simile di completezza vale per i problemi in **co-NP**.

30 – 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:

31 – 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 può 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.

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 **NP** ∩ **co-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.

32 – Esempio. Fino alla recente dimostrazione che il problema della primalità sta in **P**, si sapeva soltanto che tale problema stava in **NP** ∩ **co-NP**, in quanto banalmente il problema sta in **co-NP**, ed inoltre 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 $p-1$. Si può dimostrare (in modo alquanto laborioso) che r e i divisori primi di $p-1$ costituiscono un certificato polinomiale. ∎

33 – 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 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 16: 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). ∎

34 – 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 ricopertura di nodi. 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 ricopertura (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 ricopertura. ∎

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.

6 – Problemi NP-completi

In questa sezione vengono presentati alcuni problemi **NP**-completi insieme con le dimostrazioni della loro **NP**-completezza. 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!

35 – Esempio. Dimostriamo che il problema di decidere se il seguente insieme

$$\begin{aligned} A\,x &\geq b \\ x &\in \{0,1\}^n \end{aligned} \tag{9}$$

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 (9) 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 (9) con l'idea che un assegnamento di vero alla variabile logica corrisponde il valore 1 della variabile 0-1 e ad ogni clausola della formula una diseguaglianza di (9). 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

$$(x_1 \vee x_2 \vee \neg x_3) = V \quad \Longleftrightarrow \quad x_1 + x_2 + (1 - x_3) \geq 1 \tag{10}$$

A questo fine basta considerare un'istanza 'sì' del primo termine in (10) e far vedere che soddisfa il secondo termine e poi considerare un'istanza 'sì' del secondo termine in (10) 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 (9). 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

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \tag{11}$$

viene trasformata in

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

Come si vede, disponendo di un algoritmo che risolve (9) si è in grado di risolvere ogni istanza della Soddisfattibilità. ∎

36 – Esempio. Questo esempio di trasformazione è un po' più complesso. Si vuole dimostrare che il Problema del Massimo Insieme Indipendente (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 indipendente (nodi indipendenti) alle clausole: tante clausole soddisfatte, altrettanti nodi indipendenti. Quindi i nodi indipendenti dovranno essere almeno tanti quante le clausole se la formula è soddisfattibile. Raffinando quest'idea si può pensare che il nodo indipendente 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 indipendente per ogni insieme associato ad una clausola, 3) non avere nodi indipendenti 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 (11), le cui clausole indichiamo con A, B, C e D, genera il grafo in Figura 3, 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).

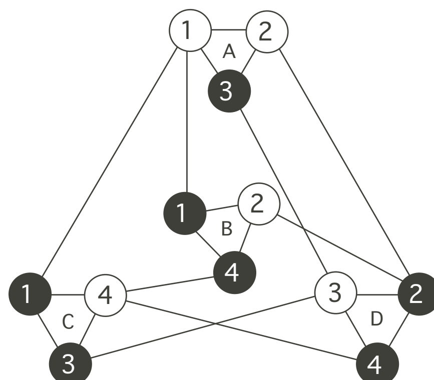


Figura 3

Costruito il grafo bisogna ora dimostrare la corrispondenza fra le istanze di tipo 'sì'. Nel problema di decisione dell'Insieme Indipendente il numero di nodi indipendenti deve essere almeno pari al numero di clausole. Si supponga sapprima che la formula booleana sia soddisfatta. Bisogna far vedere che esistono m nodi indipendenti. 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 indipendente di cardinalità m. Si è così dimostrato che ogni istanza 'sì' della Soddisfattibilità si trasforma in un'istanza 'sì' dell'Insieme Indipendente.

Bisogna ora dimostrare la stessa cosa per le istanze 'no'. È equivalente dimostrare che ogni istanza 'sì' dell'Insieme Indipendente si trasforma in un'istanza 'sì' della Soddisfattibilità. Si supponga quindi che esistano almeno m nodi indipendenti. 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 indipendenti e necessariamente uno per cricca. Se al nodo indipendente 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 indipendente possono ricevere un arbitrario valore di verità. Comunque l'assegnamento fatto con i nodi indipendenti è già sufficiente a rendere soddisfatta la formula.

Il problema dell'Insieme Indipendente è banalmente in **NP**. Quindi è **NP**-completo. Automaticamente si è dimostrata anche la **NP**-completezza della Cricca, della Ricopertura Nodi, della Ricopertura d'Insiemi e dell'Impaccamento d'Insiemi. 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).
∎

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: 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: dato un grafo G ed un intero K, esiste un taglio di G con un numero di archi non inferiore a K?
– assegnamento tridimensionale. Rimane **NP**-completo anche nella versione con le triple assegnate tramite coppie.

Sono invece risolubili polinomialmente:
– Programmazione Lineare: data una matrice A e un vettore b, esiste $\hat{x}$ tale che $A\,\hat{x} \leq b$?;
– assegnamento;
– cammino minimo in un grafo generico con costi non negativi;
– minimo albero di supporto: 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: dato un grafo G ed un numero K, esiste un taglio di G con un numero di archi non superiore a K?
– accoppiamento pesato: 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?

7 – 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 a Nicholas Pippenger). La classe **NC** si situa fra **NL** e **P**, cioè **NL** $\subset$ **NC** $\subset$ **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.

37 – 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-ma di una matrice è anche parallelizzabile in quanto la potenza k-ma viene calcolata sfruttando ricorsivamente le potenze di 2. ∎

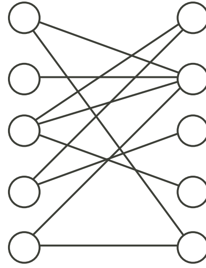


Figura 4

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

38 – Esempio. Sia dato un grafo bipartito $n \times n$, come in Figura 4 ($n = 5$) 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

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

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 $r_m \le K^{m-1} + r_{m-1}\,K$ con $r_1 \le 1$. Si ottiene facilmente $r_2 \le 2\,K$ e in generale $r_m \le m\,K^{m-1}$. 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 $\{1, 2, \ldots, 2\,m\}$ 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 $\mathbf{P} \neq \mathbf{NP}$).