UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI SCIENZE DELL'INFORMAZIONE

Dispensa del corso
di

ALGORITMI II (prima parte)

Prof. Alberto Bertoni

Anno Accademico 2011-2012

Parte A: Algoritmi e Modelli NonDeterministici e Probabilistici

1. Algoritmi non deterministici
2. La classe NP
3. Simulazione deterministica di algoritmi NonDeterministici
4. Algoritmi Probabilistici
5. La classe RP
6. Le funzioni “one-way”
7. Applicazioni: crittografia a chiave pubblica
8. Algoritmi genetici
9. Analisi elementare di algoritmi genetici
10. Algoritmi genetici come euristica per l’ottimizzazione
11. La ricerca locale
12. Ricerca randomizzata: le catene di Markov
13. Ricerca randomizzata: algoritmo di Metropolis
14. Ricerca randomizzata: annealing simulato.

Parte B: Algoritmi paralleli.

1. Modelli a memoria condivisa e a memoria distribuita
2. Modello P-RAM
3. Algoritmi su P-RAM
    - *Sommatoria*
    - *Somme Prefisse*
    - *Ordinamento bitonico*
4. Tecnica del “ciclo euleriano”
    - *Visita di alberi*
5. Reti di interconnessione
    - *Array lineare*
    - *Mesh quadrata*
    - *Albero completo*
    - *Ipercubo*

Alcuni riferimenti bibliografici

INTRODUZIONE

Il modello di calcolo dominante nella seconda metà del XX secolo è sicuramente la Macchina di Von Neumann, che ha portato a grande sviluppo i concetti riferentesi alla programmazione sequenziale. Nell'area degli algoritmi, questo ha comportato una rapida crescita di conoscenze sulle strutture dati e sulle tecniche di disegno e di analisi che si riferiscono ai cosiddetti "Algoritmi Sequenziali Deterministici", oggetto di studio del corso di "Algoritmi e strutture dati".

L'evoluzione delle tecnologie elettroniche, la spinta causata dalla tensione verso nuove applicazioni potenziali e le necessità di introdurre nuove conoscenze, in un mondo caratterizzato dalla presenza di sistemi comunicanti a distanza e in modo imprevedibile, ha richiesto di abbandonare il "paradiso terrestre" della programmazione sequenziale e di avventurarsi in nuove aree, meno sistematizzate e più rischiose. Nell'area degli algoritmi, l'introduzione di nuovi paradigmi sulla natura fisica del calcolo ha comportato attenzione allo sviluppo degli algoritmi probabilistici e quantistici; ulteriormente, le potenzialità di architetture composte da molte parti intercomunicanti ha portato alla luce la necessità di studio degli algoritmi paralleli, concorrenti e distribuiti.

Questo corso è dedicato da un lato all'introduzione di elementi sui modelli di calcolo e algoritmi probabilistici, dall'altro all'introduzione di qualche nozione sui modelli di calcolo paralleli e ai relativi algoritmi.

Gli algoritmi probabilistici sono qui introdotti come metodo per simulare realisticamente, in qualche caso, macchine non deterministiche. Il non determinismo può essere visto come un modo di descrivere problemi naturali, senza che se ne possa tuttavia dare sempre una soluzione algoritmica efficiente. Questo comporta un "gap" tra ciò che possiamo pensare e ciò che possiamo davvero fare: le probabilità possono in certi casi diminuire tale "gap" (vedremo esempi di protocolli di comunicazione in cui la probabilità diminuisce la complessità in modo sorprendente, ed esempi di algoritmi in cui la probabilità permette di testare efficientemente la primalità di un intero). Riconoscere che comunque tale "gap" rimane (faremo una discussione sulle funzioni "one-way) comporta egualmente interessanti idee algoritmiche, con applicazioni alla crittografia a chiave pubblica o alla firma digitale.

Discutiamo infine modelli probabilistici, basati su metafora naturale, che portano ad euristiche per la soluzione di un vasto insieme di problemi, dall'ottimizzazione combinatoria all'apprendimento computazionale. In questo ambito introdurremo alcune nozioni sugli algoritmi genetici (basati sull'analogia biologica e la teoria evolutiva) e sulla ricerca globale randomizzata, basata sull'analogia del processo fisico di raffreddamento.

Si è infine deciso di non introdurre in questo corso alcun elemento nell'importante area degli algoritmi distribuiti e concorrenti, ben sviluppati comunque in altri corsi dedicati a questi argomenti, centrando invece l'attenzione su alcuni aspetti del calcolo parallelo. Si discutono i due principali modelli di calcolo - a memoria concentrata (P-RAM) e a memoria distribuita (reti di interconnessione) – e si presentano algoritmi nelle varie topologie per alcuni classici problemi "benchmark", quali il MASSIMO e l'ORDINAMENTO.

In conclusione, anziché procedere in modo sistematico si è preferito in questo corso segnalare da un lato i principali problemi in alcuni settori selezionati, dall'altro procedere con esempi illustrativi, nella speranza di creare stimoli verso un'area di ricerca dai risvolti spesso imprevedibili e stupefacenti.

Parte A: Algoritmi e Modelli NonDeterministici e Probabilistici

1. ALGORITMI NON DETERMINISTICI

Durante l'esecuzione di un algoritmo deterministico, lo stato del calcolo al tempo t contiene tutte le informazioni utili a determinare in modo univoco lo stato del calcolo al tempo t+1 (*determinismo*). Ci sono tuttavia molte situazioni interessanti in cui, per determinare il nuovo stato, è richiesta informazione aggiuntiva che l'algoritmo non si può costruire: la prosecuzione del calcolo è resa possibile solo da una comunicazione dall'esterno. In questo caso, dato lo stato al tempo t, per lo stato al tempo t+1 potremo avere varie alternative, una per ogni messaggio proveniente dall'esterno: il sistema realizzato assume dunque un comportamento *non deterministico*.

Algoritmi non deterministici sono utili nella modellazione di varie situazioni, alcune delle quali saranno descritte in seguito.

Gli algoritmi saranno descritti nello pseudocodice Algol-like tipicamente usato per gli algoritmi deterministici, al cui insieme di comandi si aggiunge l'istruzione

S = CHOICE (A)

dove A assume come valori un insieme finito (di alternative).

L'esecuzione di S = CHOICE (A) ha la seguente interpretazione:

- L'algoritmo si arresta e chiede all'esterno di selezionare un elemento di A
- L'esterno seleziona liberamente (scelta non deterministica) un elemento e lo passa all'algoritmo, che puo' continuare il calcolo.

Esempio 1.1:
Un classico algoritmo nondeterministico rintracciabile su qualsiasi personal è il gioco a carte "hearts". In questo gioco l'algoritmo seleziona a caso una condizione iniziale: dopo aver eseguito una mossa, l'algoritmo si mette in stato di attesa aspettando l'alternativa che gli viene fornita dal giocatore; ricevutala, esegue una nuova mossa e questo fino a quando il gioco termina (con "successo", e allora vince il computer, o con "fallimento", e allora il computer perde). In questo caso, l'informazione necessaria alla esecuzione di S = CHOICE (A) è data dall'interazione con il giocatore.

Esempio 1.2:
In un racconto di M.Aymè, la protagonista Sabina quando vuole sedurre un nuovo amante mantenendo al contempo il precedente, genera una copia di se stessa. Ipotizzando che ad ogni intervallo di tempo ogni Sabina si raddoppi, dopo t intervalli ci saranno 2^t distinte Sabine che agiscono indipendentemente. In questo caso abbiamo dato una interpretazione del non-determinismo come parallelismo: di fronte a più alternative, l'algoritmo le segue contemporaneamente generando più copie di se

stesso. Il limite di questo approccio è la severa limitazione al numero di passi eseguibili, in presenza di naturali limiti di risorse. Per esempio, dopo soli 57 intervalli di tempo ci sarà sulla terra una Sabina per metro quadro!

Esempio 1.3:
Un semplice modello di apprendimento on-line può essere descritto mediante interazione tra uno studente e un insegnante.
Lo studente deve individuare, all'interno di una famiglia di concetti (per esempio, i triangoli nel piano) un particolare concetto (un triangolo) sulla base di *esempi* forniti dall'interazione con un insegnante.
Dato un triangolo g, un "esempio" è descritto da una coppia (x,g(x)), dove x è un punto del piano e, se x sta in g, allora g(x)=1, altrimenti g(x)=0. Diremo "positivo" un esempio di tipo (x,1) e negativo quello di tipo (x,0).

L'insegnante ha a disposizione un insieme E di esempi di un concetto g:
$E = \{(x_1, g(x_1)), \ldots, (x_m, g(x_m))\}$.
Obiettivo dello studente è quello di "*imparare*" dall'interazione con l'insegnante un concetto h che, pur non coincidendo necessariamente con g, classifica correttamente tutti gli esempi, cioè tale che:

$$g(x_k) = h(x_k) \qquad (\mathrm{k} = 1, \mathrm{m})$$

L'interazione on-line tra studente e insegnante può essere modellata dal seguente schema di algoritmo non-deterministico

Algoritmo di APPRENDIMENTO:

IPO = un concetto b *{ipotesi iniziale}*

While "un esempio in E non e' classificato correttamente da IPO" do

$$\begin{cases} (\mathrm{x}, \mathrm{y}) = \mathrm{CHOICE}\,(E) \\ \text{if } \mathit{IPO(x)} \neq y \quad \text{do} \quad \mathit{IPO} = \text{Aggiorna}(\mathit{IPO}, (\mathrm{x}, \mathrm{y})) \end{cases}$$

Gli algoritmi di apprendimento, in questo modello, differiscono a causa della diversa scelta delle ipotesi iniziale e dal diverso meccanismo di aggiornamento. Un famoso algoritmo è il cosiddetto Percettrone [Rozenblatt, 1956], che "impara" un semispazio separando gli esempi positivi da quelli negativi con un iperpiano.

Esempio 1.4:

Problema Clique

Il problema ‘Clique’ può essere così enunciato:

Istanza:

- Un grafo non orientato G = < V, E > con vertici V e lati E
- Un intero k

Questione: esiste una cricca (= sottografo completo) di G di k vertici? (un sottografo completo è descritto da un sottoinsieme di vertici tutti collegati uno all’altro da lati in E)

Un algoritmo non deterministico per ‘Clique’ è il seguente:

Input: (< V, E >, K)
$C = \phi$; $A = V$
Do
$S = CHOICE(A)$
$C = C \cup \{S\}$
$A = A - \{S\}$
Until “C è una cricca”
If $|C| \geq K$ *Then Successo*
Else Fallimento

Chiaramente, comunque vengano eseguite le scelte eseguendo l’istruzione nondeterministica S = CHOICE(A), l’algoritmo determina un sottografo completo massimale; se questo sottografo ha almeno k elementi allora il risultato è “Successo”, altrimenti “Fallimento”. Possiamo osservare:

- Se il grafo G = < V, E > ha una cricca di dimensione k, allora esiste una sequenza di scelte inviate dall’esterno per cui l’algoritmo termina con “Successo”.
- Se il grafo G = < V, E > ha non una cricca di dimensione k, allora per qualsiasi sequenza di scelte inviate dall’esterno l’algoritmo termina con “Fallimento”.

Osserviamo inoltre che, se il grafo G = < V, E > ha una cricca di dimensione k, il numero di scelte che porta a successo è al più |V| e la computazione che porta a successo richiede al più O(|V|) passi di calcolo.

Esempio 1.5:

Model Checking

Il problema del Model Checking può essere così enunciato:

Istanza:

- Una struttura relazionale finita a (data da un insieme A e da relazioni di varia arietà $R_1,...,R_s$)
- Una formula $\varphi(x_1,....,x_s)$ in qualche logica L

Risposta: E' φ soddisfattibile in a ? (E' cioè possibile trovare elementi $a_1,....,a_s$ in a per cui $\varphi(a_1,....,a_s)=1$)

Il problema precedente è molto generale. Si osserva che quando $\varphi(x_1,....,x_s)$ è una formula del 1° ordine (oppure rappresentabile in un linguaggio commerciale come SQL) il problema diventa quello di interrogare un data-base.

Qui il problema è formulato come problema di decisione. Sono possibili formulazioni diverse, come quella di *ricerca* (determinare un assegnamento alle variabili che soddisfa la formula) o di *enumerazione* (elencare tutti gli assegnamenti che soddisfano la formula).

Osserviamo che 'Clique' è un caso particolare di Model Checking:

1. La struttura relazionale è data da un insieme (A=V) e da una relazione binaria E (E(x,y) = if x,y sono collegati da un lato in E then 1 else 0)
2. La formula $\varphi(x_1,....,x_k)$ è la seguente:

$$\bigwedge_{\substack{1\le i<l\le k \\ i\neq l}} E(x_i,x_l) \wedge \bigwedge_{\substack{1\le i<l\le k \\ i\neq l}} x_i \neq x_l$$

Infatti, un eventuale assegnamento $(a_1,....,a_k)$ che soddisfa tale formula deve essere tale che $E(a_i,a_l)=1$ con $a_i \neq a_l$ (i=1,k e l=1,k), e quindi l'insieme $a_1,....,a_k$ forma una cricca di k vertici.

In Esempio 1.1 è l'abilità del giocatore (esterno) che porta al successo; naturalmente questo è possibile quando almeno una sequenza di calcolo porta al successo.

Esempio 1.2 interpreta il non determinismo come una sorta di parallelismo, coi limiti prima accennati.

Nel modello di apprendimento di Esempio 3, è importante introdurre un parametro di valutazione, che ci permetta di dire quando un algoritmo è "migliore" di un altro. Osserviamo ora che, a causa della condizione nel ciclo "while", quando una sequenza di calcolo termina l'ipotesi ottenuta classifica correttamente tutti gli esempi: una

naturale misura della ‘bontà’ di un algoritmo è data allora dal massimo numero di esempi mal classificati in una qualsiasi sequenza di calcolo che porti a terminazione.

In Esempio 1.4, se il grafo < V, E > ammette sottografi completi con K vertici, c’è una sequenza di scelte “libere” che, passata all’algoritmo non deterministico, genera una computazione che porta a successo; viceversa, se < V, E > non ammette sottografi completi con K vertici nessuna sequenza di calcolo dell’algoritmo nondeterministico può portare a successo. Questo problema, così come il problema Model Checking, può essere visto come prototipo dei problemi della classe NP.

2. LA CLASSE NP

Una importante classe di problemi "trattabili" con algoritmi deterministici è la classe P. Un problema è nella classe P se può essere risolto da un algoritmo deterministico che, su istanza I, genera una sequenza di calcolo di al più p(|I|) passi, dove |I| è la dimensione dei dati relativi all'istanza I e p(x) è un opportuno polinomio. L'importanza di questa classe è legata a due motivi:

- Robustezza. Dati vari modelli di calcolo (ad esempio: macchine di Turing a un nastro, a più nastri, macchine RAM), le classi di problemi risolubili in tempo polinomiale nei vari modelli vengono a coincidere. La classe P risulta allora indipendente dal particolare formalismo usato per esprimere gli algoritmi.
- Tesi di Church estesa. Secondo questa tesi, questa classe contiene i problemi "praticamente risolubili" (si veda per una discussione il classico libro "Computers and Intractability" di Garey-Johnson).

Una classe che contiene invece molti problemi di interesse pratico è la classe NP. Essenzialmente, essa contiene problemi per cui è computazionalmente difficile trovare una soluzione ma, data una potenziale soluzione, è computazionalmente facile effettuarne la verifica. L'eventuale esistenza di problemi in NP ma non in P può essere letta come un gap tra le aspirazioni dell'uomo e la sua capacità di risolverle.

Allo scopo di introdurre formalmente NP, richiamiamo che l'esecuzione di un algoritmo non deterministico (che chiameremo "VERIFICATORE") richiede l'interazione con l'esterno (che chiameremo "DIMOSTRATORE"): ogni qualvolta debba essere eseguita l'istruzione S=CHOICE(A), il DIMOSTRATORE invia l'informazione necessaria (un elemento scelto in A) al VERIFICATORE; avendo tali informazioni, il VERIFICATORE può continuare l'esecuzione. Naturalmente il DIMOSTRATORE può inviare la parola contenente tutte le informazioni prima dell'inizio del calcolo, e il VERIFICATORE può, sulla base di tali informazioni, eseguire il calcolo senza più interagire. Se il calcolo porta a successo, la parola contenente le informazioni sarà detta "testimone di successo".

Un algoritmo nondeterministico può allora essere visto come l'interazione tra un VERIFICATORE e un DIMOSTRATORE descritta dal seguente protocollo.

Avendo in ingresso l'istanza I del problema da risolvere, il VERIFICATORE richiede informazioni al DIMOSTRATORE; il DIMOSTRATORE passa al VERIFICATORE tali informazioni sotto forma di una parola W; avendo ora in ingresso I e W, il VERIFICATORE può eseguire il calcolo. Vale inoltre che:

- Se I verifica il problema, il DIMOSTRATORE può inviare un testimone di successo, cioè una parola W che induce una computazione che porta a successo.
- Se I non verifica il problema, qualsiasi parola invii il DIMOSTRATORE induce una computazione che porta a fallimento.

Un problema è in NP se esiste un algoritmo nondeterministico per cui, quando l'istanza I verifica il problema, la dimensione |W| del testimone di successo è un polinomio della dimensione |I| dell'istanza, ed inoltre anche la lunghezza del calcolo che porta a successo è un polinomio della dimensione |I| dell'istanza.

Esempio 2.1:
Il problema Clique sta in NP; infatti, dato il grafo G, se c'è una cricca W di dimensione k, il DIMOSTRATORE può inviare tale cricca come testimone i successo e il VERIFICATORE può controllare che tutti il lati di W sono effettivamente lati del grafo. La dimensione |W| del testimone è essenzialmente k, che è al più uguale al numero n di vertici del grafo G, assunta come dimensione di G; inoltre, la verifica che tutti il lati di W sono effettivamente lati del grafo richiede al più n^2 passi.

Esempio 2.2:
Una formula booleana $F(x_1,.., x_n)$ è una tautologia se $F(a_1,.., a_n) = 1$ per ogni assegnamento $(a_1,.., a_n)$, è una contraddizione se $F(a_1,.., a_n)= 0$ per ogni assegnamento $(a_1,.., a_n)$, infine è una formula "anfotera" se esiste un assegnamento $(a_1,.., a_n)$ per cui $F(a_1,.., a_n) = 1$ e un assegnamento $(b_1,.., b_n)$ per cui $F(b_1,.., b_n) = 0$. Il problema TAUTOLOGIA richiede, data come istanza una formula booleana F, di decidere se F è una tautologia; il problema FORMULA ANFOTERA richiede invece di decidere se F è anfotera.

Il problema FORMULA ANFOTERA è in NP: data infatti l'istanza $F(x_1,.., x_n)$, se F è anfotera il DIMOSTRATORE invia gli assegnamenti $(a_1,.., a_n)$ e $(b_1,.., b_n)$ per cui cui $F(a_1,.., a_n) = 1$ e $F(b_1,.., b_n) = 0$. Tali assegnamenti hanno dimensione minore di quella della formula, ed inoltre il VERIFICATORE, avendo in ingresso F, $(a_1,.., a_n)$ e $(b_1,.., b_n)$, può verificare in tempo lineare che $F(a_1,.., a_n) = 1$ e $F(b_1,.., b_n) = 0$.

Non si vede invece come TAUT possa essere in NP. Se l'istanza F è una tautologia, il DIMOSTRATORE non può usare la tabella di verità di F come testimone di successo, perché tale tabella, contenendo 2^n assegnamenti, può risultare di dimensione esponenziale rispetto a quella della formula. Il DIMOSTRATORE potrebbe in alternativa prendere in considerazione un calcolo logico ed inviare come testimone la dimostrazione W, data nel calcolo logico, che F è una tautologia. In questo caso il VERIFICATORE può, avendo in ingresso F e W, effettuare la verifica in tempo essenzialmente $O(|F|+|W|)$. Il problema qui è che, almeno nei calcoli logici fino ad ora noti, esistono infinite tautologie F, per cui la dimostrazione W più corta è esponenzialmente lunga rispetto a $|F|$!

Esempio 2.3:
Il problema COMPOSTO ha come istanza la rappresentazione binaria $x_1..x_n$
di un intero x, e la questione richiede di decidere se x è composto, se cioè esiste una fattorizzazione $x=a.b$ non banale (cioè con $1<a,b<x$).

COMPOSTO è chiaramente in NP: è infatti sufficiente che il DIMOSTRATORE, se x è composto, invii come testimoni le rappresentazioni binarie di a e b per cui $a.b=x$; tali rappresentazioni richiedono al più 2n bit, inoltre il VERIFICATORE, avendo in ingresso x,a,b, può verificare che $x=a.b$ in tempo al più n^2, usando l'algoritmo di moltiplicazione imparato alle scuole elementari (o più sofisticati algoritmi che richiedono $O(n.\lg n. \lg\lg n)$ passi).

Esempio 2.4:
Il problema PRIMO ha come istanza la rappresentazione binaria $x_0..x_{n-1}$ di un intero x di n bit, e la questione richiede di decidere se x è primo, cioè se ogni fattorizzazione $x=a.b$ implica che $a=1$ o $b=1$.

Chiaramente un numero x è primo se, per ogni intero y ($1<y<1+x^{1/2}$), x non è divisibile per y. Naturalmente il DIMOSTRATORE non può inviare come testimone di primalità i possibili resti della divisione tra x e y ($1<y<1+x^{1/2}$), poiché il numero di tali resti è $2^{n/2}$, un esponenziale nella dimensione n di x.

Possiamo tuttavia osservare che i numeri primi sono caratterizzati dalla seguente proprietà:

Fatto: p è primo se e solo se esiste un intero t ($1<t<p$) tale che:

- $\langle t^{(p-1)} \rangle_p = 1$
- $\langle t^{(p-1)/q} \rangle_p \neq 1$ per ogni divisore primo q di p-1

Questo permette, come osservato da Pratt nel 1975, di mostrare che il problema PRIMO sta in NP. Il protocollo è il seguente.

1. Se x è pari, il VERIFICATORE decide che x è primo solo se $x=2$, senza bisogno di interagire col DIMOSTRATORE.
2. Se x è dispari (ed è primo), allora il VERIFICATORE richiede la prova al DIMOSTRATORE e riceve come testimone di primalità l'informazione $W(x)$, costituita dal numero t di *Fatto*, dai divisori primi $q_1,..,q_s$ di x-1che sono maggiori di 2 e, ricorsivamente, dai testimoni $W(q_1),\dots,W(q_s)$ di primalità dei numeri $q_1,..,q_s$:

$$W(x) = \langle x,t,q_1,..,q_s , W(q_1),\dots,W(q_s)\rangle$$

Il VERIFICATORE, conoscendo $x,t,q_1,..,q_s$, verifica $\langle t^{(x-1)} \rangle_x = 1$ e che $\langle t^{(x-1)/qk} \rangle_x \neq 1$; conoscendo inoltre $W(q_1),\dots,W(q_s)$ verifica che $q_1,..,q_s$ sono primi e, mediante divisioni successive, controlla che $q_1,..,q_s$ sono i soli divisori primi dispari di x-1.

Per dimostrare che il problema è in NP, indichiamo con $Z(x)$ il numero complessivo di primi inviati dal DIMOSTRATORE al VERIFICATORE e proviamo che $Z(x) \leq (\log x)^2 \leq n^2$.

Per prima cosa, osserviamo che, se $q_1,..,q_s$ sono i divisori primi dispari di x, vale:

$$\text{(a)} \quad Z(x) \leq s + Z(q_1) + \dots + Z(q_s)$$

Ricordando inoltre che $x > x-1 > q_1...q_s > 2^s$, risulta:

$$\text{(b)} \quad n \geq \log x > \log q_1 + \dots + \log q_s > s$$

(c)

Grazie ad (a) e (b), dimostriamo ora per induzione che $Z(x) \leq (\log x)^2$.

1. Se $x=2$ allora $Z(x)=0<1$
2. Proviamo che la proprietà vale per x, supponendola vera per tutti i numeri $<x$:

$$\begin{aligned} Z(x) &\leq s + Z(q_1) + \dots + Z(q_s) && \text{(per (a))} \\ &\leq s+(\log q_1)^2 +(\log q_2)^2 + \dots +(\log q_s)^2 && \text{(ipotesi di induzione)} \\ &< \log x+(\log q_1)^2 +(\log q_2 + \dots +\log q_s)^2 \\ &< \log x +(\log q_1)^2 +(\log x -\log q_1)^2 && \text{(per (b))} \\ &= \log x +(\log x)^2 -2(\log x -\log q_1) .(\log q_1) \\ &\leq \log x +(\log x)^2 -2(\log x -1) \end{aligned}$$

L'ultima riga deriva dal fatto che $q_1 \geq 2$ e che $2(\log x -\log q_1) .(\log q_1)$ raggiunge il minimo per $q_1=2$. Possiamo concludere:

$$Z(x) < \log x +(\log x)^2 -2(\log x -1) = (\log x)^2 - \log x +2 \leq (\log x)^2$$

Per provare che PRIMO è in NP, basta osservare:

- Il testimone W(x) ha una dimensione $O(n^3)$, polinomiale rispetto alla dimensione $|x|=n$. Infatti, W(x) contiene $Z(x)= n^2$ numeri primi, ognuno dei quali ha associato numeri che richiedono al più 2n bit (gli analoghi dei numeri $t,q_1,..,q_s$).
- Il tempo di calcolo dell'algoritmo VERIFICATORE su ingresso $p,t,q_1,..,q_s$, è un polinomio in n. Basta utilizzare la seguente procedura, che calcola $< a^b >_c$ per "quadrature successive":

Procedura MODPOT(a, b, c)
if b=1 then return $< a >_c$
else if (b pari) then $z = MODPOT(a, \frac{b}{2}, c)$
return $< z^2 >_c$
else $z = MODPOT(a, b-1, c)$
return $< a \cdot z >_c$

Questa procedura calcola $< a^b >_c$ mediante O(lg b) prodotti di numeri di log c bit. Nel nostro caso, lg c , lg b $\leq$ n, quindi il tempo di esecuzione della procedura è $O(n^3)$. Poiché la procedura viene eseguita $O(n^3)$ volte, il tempo complessivo è $O(n^6)$.

Il risultato di Pratt è attualmente superato dal fatto che M. Agraval, N. Kayal, N. Saxena (AKS) che nel 2002 hanno esibito un algoritmo deterministico che risolve il problema PRIMO in tempo polinomiale $O(n^{12})$.

3. SIMULAZIONE DETERMINISTICA DI ALGORITMI NON DETERMINISTICI

Ogni procedura non deterministica può essere simulata da una procedura deterministica equivalente. Mostriamo in questa sezione dapprima uno schema derivato dalla tecnica enumerativa e poi, segnatamente per i problemi di ottimizzazione, lo schema cosiddetto Branch&Bound.

1 – Schema enumerativo

Studiamo qui come si possa simulare una procedura non deterministica con una deterministica, rilevando al contempo come questa soluzione per molte applicazioni risulti impraticabile a causa dell'alto tempo di esecuzione.

Per semplicità,

1. Ci limitiamo a considerare problemi di decisione.

   Un algoritmo non deterministico *A* risolve un dato problema di decisione *P* se:
   - Su istanza 'X' la risposta di P è positiva $\Rightarrow$ almeno una sequenza di calcolo di *A* su ingresso X ha successo.
   - Su istanza 'X' la risposta di P è negativa $\Rightarrow$ tutte le sequenze di calcolo di *A* su ingresso X hanno fallimento.

2. Supponiamo che ogni sequenza di calcolo incontri l'istruzione 'CHOICE' al più *N* volte; indicheremo in seguito con 'S[k]' la scelta operata sull'insieme 'A' nell'esecuzione di 'CHOICE(A)' la k^a volta (k=1,N).

Sotto questa ipotesi, uno schema generale di procedura non deterministica è la seguente:

Procedura NON DETERMINISTICA (...)

$$\text{For } K=1,N \text{ do } \begin{cases} \text{Seleziona A in funzione di } S[1],...,S[k-1] \\ \\ \text{if } A=\phi \text{ then FAILURE} \end{cases}$$

$$\text{else} \begin{cases} S[K]=CHOISE(A) \\ \\ \text{if "} S[1],...,S[K] \text{ porta a soluzione" then SUCCESS} \end{cases}$$

Le possibili sequenze di computazione della Procedura NON DETERMINISTICA su un dato ingresso sono descritte da un albero; per simulare tale procedura con una deterministica, basterà visitare l'albero finchè non si trova una foglia di 'successo'; nel caso non esistano foglie di 'successo', la procedura deterministica dopo la visita di tutto l'albero dichiarerà 'fallimento'.
In seguito indicheremo con $|A|$ la cardinalità di A e con a[k] il k-esimo elemento in A (k=1, $|A|$).

Chiave della simulazione è la seguente procedura, che per semplicità diamo in forma ricorsiva:

Procedura ENUMERAZIONE (k: intero)
A=insieme selezionato da S[1],…,S[k-1]

If $k \leq N$ *then*
If $A \neq \phi$ *then*

$$for\ j=1,\ |A|\ do \begin{cases} S[k]=a[j] \\ \\ if\ "S[1]....S[k]\ porta\ a\ soluzione"\ then\ SUCCESS \\ else ENUMERAZIONE(k+1) \end{cases}$$

La simulazione si ottiene invocando:

'ENUMERAZIONE (1)'

Supponiamo che ogni computazione dell'algoritmo non deterministico abbia tempo di calcolo ≤ T_{ND} e che l'insieme di alternative A sia tale che $|A| \leq m$. Ricordando che un albero m-ario di altezza N ha al più m^N foglie, si ha:

Fatto 3.1: Sia T_D il tempo di esecuzione della procedura Enumerazione.
Allora: $T_D \leq T_{ND}\ m^N$

Si osservi che, per $m \geq 2$, la crescita della funzione m^N è rapidissima. In gran parte delle applicazioni di interesse pratico, se n è la dimensione dell'ingresso, sia T_{ND} che m e N sono polinomi in n e questo porta a tempi di calcolo T_D esponenziali nella dimensione dell'ingresso; l'algoritmo deterministico ottenuto può allora rispondere in tempi ragionevoli solo su dati di piccola taglia.

2 – Schema branch&bound

Una importante classe di problemi la cui soluzione esatta può essere ottenuta trasformando in procedura deterministica un algoritmo non deterministico è quella dei problemi di ottimizzazione combinatoria. Un *problema di ottimizzazione* è descritto assegnando (spesso implicitamente) una famiglia A di soluzioni ammissibili e una funzione obbiettivo $f : A \rightarrow R$; il problema richiede di determinare il massimo (o il minimo) globale di f:

$$a = \text{argmax}_{x \in A} f(x)$$

In un problema di *ottimizzazione combinatoria* le soluzioni ammissibili possono essere descritte attraverso un algoritmo non deterministico: qui supporremo per semplicità che ogni soluzione ammissibile sia ottenuta da una sequenza di calcolo che richieda lo stesso numero n di scelte S[1], S[2],.., S[n].

Esempio 3.1: Max Cut.
Dato un grafo $G = \langle V,E \rangle$, le soluzioni ammissibili sono i sottoinsiemi di V. Il peso f(X) di una soluzione ammissibile $X \subseteq V$ è il numero di lati che hanno un estremo in X e l'altro nel suo complemento X^c. Il problema richiede di determinare argmax f(X).
Un algoritmo non deterministico che descrive le soluzioni ammissibili è il seguente:

```
Input (N)
S = Φ
For k=1,N  do
        S[k] = CHOICE (T,F)
        If S[k] = T then  S = S∪{k}
```

Esempio 3.2: Commesso Viaggiatore
Data una matrice nxn a componenti d(i,k) intere non negative, le soluzioni ammissibili sono permutazioni P = (P[1],…, P[n]) degli interi {1,2,…,n}, Il peso f(P) di una soluzione ammissibile P è dato da:

$$f(P)=d(P[1], P[2])+ \dots +d(P[N-1], P[N])$$

Il problema richiede di determinare $S = \text{argmin}_P \; f(P)$.
Un algoritmo non deterministico che descrive le soluzioni ammissibili è il seguente:

```
Input (N)
A = {1,2,...,N}
For k=1,N do
        P[k] = CHOICE (A)
         A    = A - { P[k] }
```

Un algoritmo deterministico “forza bruta” che permette di risolvere il problema di ottimizzazione può essere ottenuto direttamente dalla procedura ENUMERAZIONE, che enumera tutte le soluzioni ammissibili, semplicemente calcolando il peso di ogni soluzione ammissibile e memorizzando la migliore trovata. Come abbiamo visto, questo approccio in generale porta a tempi di calcolo elevati anche per dati di modesta dimensione. Può essere allora interessante sviluppare tecniche che portano a riduzioni del tempo di calcolo: gli algoritmi Branch&Bound rappresentano una di queste tecniche che, pur non ottenendo in generale tempi di calcolo polinomiali nella dimensione dell’ingresso, forniscono spesso prestazioni accettabili.

Per illustrare tale tecnica, consideriamo un problema di minimizzazione le cui soluzioni ammissibili sono ottenute con un algoritmo non deterministico; supponiamo inoltre che ogni soluzione ammissibile può essere ottenuta con N scelte S[1], S[2],.., S[N]. Date le prime k scelte scelte S[1], S[2],.., S[k] (con k eventualmente 0), diremo *I*(S[1], S[2],.., S[k]) l’insieme delle scelte da cui estrarre scelte S[k+1], e *A*(S[1], S[2],.., S[k]) l’insieme delle soluzioni ammissibili ottenibili aggiungendo nuove scelte al segmento iniziale S[1],.., S[k].

Supponiamo inoltre di conoscere una funzione L(S[1], S[2],.., S[k]) tale che:
1 - L sia calcolabile in modo efficiente
2 - L(S[1], S[2],.., S[k]) ≤ $\text{Min}_{S\in A(S[1], S[2],.., S[k])}$ f(S) , in modo che
L(S[1], S[2],.., S[k]) risulti minorante dei valori assunti dalla funzione obbiettivo sulle soluzioni ammissibili ottenute aggiungendo nuove scelte a S[1], S[2],.., S[k].

L’idea base della procedura Branch&Bound è che se conosciamo una soluzione ammissibile S con f(S) ≤ L(S[1], S[2],.., S[k]), allora possiamo evitare di esplorare soluzioni ammissibili in *A*(S[1], S[2],.., S[k]), poiché se S’ ∈*A*(S[1], S[2],.., S[k]), allora possiamo garantire che f(S) ≤ f(S’).Lo schema generale della procedura Branch&Bound è descrivibile da una procedura ricorsiva con parametro i (1≤ i ≤ n) e due variabili globali Minsol e Mincost:

```
Procedura Branch&Bound (i)
For j=1,card I(S[1], S[2],.., S[i-1]) do
     S[i] = j° elemento in I(S[1], S[2],.., S[i-1])
     if L(S[1], S[2],.., S[i]) ≤ Mincost  then  if  i<N  then Branch&Bound (i+1)
                                                  if i =N  then  S = sol amm. data da da S[1],.., S[n]
                                                                 if f(S)<Mincost  then Minsol  = S
                                                                                       Mincost = f(S)
```

Un algoritmo deterministico che risolve il problema di ottimizzazione con la tecnica Branch&Bound può allora essere il seguente:

Minsol = una soluzione ammissibile ottenuta “ad hoc”
Mincost = f(Minsol)
Procedura Branch&Bound (1)

4. ALGORITMI PROBABILISTICI

Durante l'esecuzione di un algoritmo non deterministico, l'algoritmo si arresta a fronte dell'esecuzione CHOICE(A), richiedendo all'esterno la selezione tra le varie alternative.

Un modo per poter continuare l'esecuzione senza richiedere informazioni all'esterno è quello di scegliere a caso una delle alternative. Questo può essere fatto sostituendo l'istruzione S=CHOICE(A) con l'istruzione S= RANDOM(A).

Il significato di

S= RANDOM(A)

è quello di assegnare ad S un elemento estratto a caso da A, con distribuzione uniforme.

Otteniamo quindi una simulazione probabilistica dell'algoritmo non deterministico, che equivale a scegliere a caso una sequenza di calcolo nell'albero delle computazioni non deterministiche, producendo in uscita il valore ottenuto dall'esecuzione della sequenza. Chiameremo quindi *probabilistici* (o *stocastic*i, o *randomizzati*) algoritmi descritti nello pseudocodice Algol-like usato per gli algoritmi deterministici, al cui insieme di comandi si aggiunge l'istruzione S= RANDOM(A), la cui semantica è stata precedentemente descritta.

Un algoritmo deterministico è utilizzato per calcolare la funzione f(x) che associa al valore di ingresso x il valore di uscita y=f(x). E' possibile estendere tale nozione al caso probabilistico? Il problema qui è che diverse esecuzioni dell'algoritmo possono portare a risultati differenti. Tuttavia la seguente definizione è consistente:

- Un algoritmo probabilistico calcola una funzione f(x) se, avendo come ingresso x, dà in uscita f(x) con probabilità almeno 3/4.

La definizione è ben posta, perché è impossibile avere in uscita due diversi valori, ognuno dei quali ha probabilità 3/4 (poiché 3/4+3/4>1!). Si osservi inoltre che, nella simulazione probabilistica di un algoritmo non deterministico, il tempo di calcolo è limitato dal tempo non deterministico T_{ND} , del tutto realistico in molte applicazioni .

Osserviamo che l'esecuzione di un algoritmo probabilistico che calcola la funzione f(x) non dà, su ingresso x, necessariamente f(x): l'algoritmo può pertanto commettere errore. Quel che possiamo dire è, tuttavia, che la probabilità d'errore è limitata a 1/4:

- Prob{su ingresso x l'algoritmo risponde $z \neq f(x)$} $\leq$ 1/4

Un parametro importante dell'algoritmo probabilistico è allora la sua probabilità d'errore: essa quantifica il rischio che accettiamo nell'usare tale algoritmo.

Sia ALG un algoritmo probabilistico che calcola la funzione f(x) con probabilità d'errore 1/4. Un modo per costruire un algoritmo probabilistico che calcola la stessa funzione f(x) con una probabilità di errore inferiore a 1/4 è quello di ripetere in maniera indipendente, su ingresso x, l'algoritmo un certo numero di volte t e dare in uscita la risposta che si presenta con maggior frequenza:

```
Input(x)
  For k=1,t do y[k] = ALG(x)
  y = elemento più ripetuto in y[1], y[2], … ,y[t]
  return(y)
```

La probabilità d'errore è maggiorata dalla probabilità che, su t ripetizioni indipendenti di ALG, la frequenza di f(x) nelle risposte y[1], y[2], … ,y[t] sia inferiore a 1/2. Poiché ci si aspetta che la frequenza di f(x) in y[1], y[2], … ,y[t] si concentri intorno a 3/4, la probabilità che la frequenza di f(x) in y[1], y[2], … ,y[t] sia inferiore a 1/2 va rapidamente a 0 quando t va a ∞.

Per ottenere una misura quantitativa della probabilità di errore, cominciamo con l'osservare che la frequenza di f(x) in y[1], y[2], … ,y[t] è una variabile aleatoria F distribuita come una binomiale B(t,3/4). Per una variabile aleatoria X distribuita come una binomiale B(t,p) vale la seguente disuguaglianza di Chernoff:

$$\text{Probabilità } (X/t \leq x) \leq e^{-2t.(x-p)\hat{}2}$$

Nel nostro caso, p=3/4 e x=1/2, quindi:

$$\text{Probabilità di errore} \leq \text{Probabilità } (F/t \leq 1/2) \leq e^{-t/8}$$

Se noi siamo disponibili ad accettare il rischio di una probabilità d'errore δ, basta ripetere t volte l'algoritmo ALG con t tale che $\delta = e^{-t/8}$, cioè $t = 8.\ln 1/\delta$.

Esempio 4.1: Protocollo di comunicazione randomizzato per l'uguaglianza.
Due ricercatori R1 ed R2 sviluppano a distanza e indipendentemente sui loro computer due data base sullo stesso soggetto (per esempio una sequenza genomica). Al termine dello sviluppo, il data base di R1 conterrà la parola binaria $a = a_1,.., a_n$, quello di R2 la parola binaria $b = b_1,.., b_n$. Al solito, a e b possono essere interpretati come interi in rappresentazione binaria. I due ricercatori vogliono ora controllare se i due dati, come ci si aspetta, sono uguali. Questo può essere fatto col seguente protocollo di comunicazione:

- *R1 invia* $a = a_1,.., a_n$ *a R2*
- *R2 riceve* $a = a_1,.., a_n$ *da R2 e confronta a con b: se a=b risponde "successo" altrimenti risponde "fallimento".*

Osserviamo che il protocollo in questione è deterministico, e prevede lo scambio di n+1 bit. Tale protocollo è ottimale: è possibile provare che non ci sono protocolli deterministici corretti che prevedono lo scambio di meno di n bit.

La situazione cambia radicalmente se ammettiamo algoritmi randomizzati. Un possibile protocollo probabilistico è il seguente:

- *R1 calcola* $s=RANDOM(\{p|\ p \text{ primo}, 2\leq p\leq n^2\});\ z=<a>_s$
- *R1 invia a R2 la coppia* $<s,z>$
- *R2 riceve da R1 la coppia* $<s,z>$ *e calcola* $<b>_s$; *se* $<b>_s=z$ *allora risponde "successo" altrimenti risponde "fallimento".*

Quanti bit R1 invia a R2? Poiché $z<s\leq n^2$, sia s che z sono esprimibile con al più $\log n^2 = 2\log n$ bit: R1 invia dunque $4\log n$ bit a R2.

Il protocollo è probabilistico, quindi deve essere prevista la possibilità di errore. Qual è la probabilità d'errore? E' facile convincersi che l'unico caso di errore si ha quando $a\neq b$ ma $<a>_s = <b>_s$. Senza perdita di generalità, supponiamo che a>b: posto a-b=v, sarà $v<2^n$, $v\neq 0$ ma $<v>_s =0$. Ricordiamo che:

Probabilità di errore = #Casi favorevoli / #Casi possibili

dove: Casi possibili = $\{s|\ s \text{ primo}, 2\leq s\leq n^2\}$
Casi favorevoli = $\{s|\ s \text{ primo}, 2\leq s\leq n^2, s \text{ divide } v\}$

Poiché il numero di primi minori o uguali a N è approssimativamente N/lnN, vale:

#Casi possibili $\approx n^2/\ln n^2$.

Il numero di casi favorevoli è il numero di divisori primi di v; decomponendo v in fattori primi si ha: $2^n >v= p_1^{j1}\dots p_t^{jt} > 2^t$, e quindi il numero t di fattori primi di v è al più n:

#Casi favorevoli < n .

Possiamo concludere che la probabilità d'errore è al più 2 ln n/n.

Esempio 4.2: algoritmo probabilistico per decidere la primalità.

Presentiamo qui un classico algoritmo probabilistico, dovuto a Miller-Rabin, che decide se un intero $x = x_1,.., x_n$ rappresentato in notazione binaria è o non è primo. Esso riposa sulle seguenti considerazioni aritmetiche:

- Consideriamo un primo $p>2$. E' noto che Z_p, la classe dei resti modulo p con somma e prodotto modulo p, è un campo. In tale campo l'equazione $x^2=1$ ammette le sole soluzioni $x=1$ e $x=-1$, altrimenti sarebbe $<(x+1)(x-1)>_p = 0$ e quindi $x+1$ e $x-1$ sarebbero divisori di p.
- Per ogni a ($1 \leq a < p$) si ha che $<a^{p-1}>_p = 1$. Questo fatto è noto come "piccolo teorema di Fermat", e dipende dal fatto che, se il gruppo ciclico moltiplicativo generato da a è di ordine g, allora $<a^g>_p = 1$; inoltre tale gruppo è sottogruppo del gruppo moltiplicativo Z_p - $\{0\}$, che contiene p-1 elementi. Allora per il teorema di Lagrange g è un divisore di p-1, cioè $p-1=h.g$, e quindi $<a^{p-1}>_p = <a^{h.g}>_p = <(a^g)^h>_p = <(1)^h>_p = 1$.

Dato il primo $p>2$, p-1 è pari quindi può essere decomposto come $p-1 = 2^s.d$, dove d è dispari e $s>0$. Fissato un intero a ($1 \leq a < p$), si ha che $<a^{p-1}>_p = 1$, cioè che $<a^{2\wedge s.d}>_p = <(a^{2\wedge(s-1).d})^2>_p = 1$. Segue che $a^{2\wedge(s-1).d}$ è una radice quadrata, e quindi $<a^{2\wedge(s-1).d}>_p = -1$ o $<a^{2\wedge(s-1).d}>_p = 1$; iterando il ragionamento precedente concludiamo:

se p è primo *allora* o $<a^d>_p = 1$ o $<a^d>_p = -1$ o $<a^{2.d}>_p = -1$ o … o $a^{2\wedge(s-1).d}>_p = -1$.

Passando alla proposizione contronominale:
se $<a^d>_p \neq 1$ e $<a^d>_p \neq -1$ e $<a^{2d}>_p \neq -1$ e … e $<a^{2\wedge(s-1).d}>_p \neq -1$ *allora* p è composto

Questo giustifica il seguente algoritmo probabilistico:

Algoritmo MILLER-RABIN *(un intero $x = x_1,.., x_n$ in notazione binaria)*

- *Scrivi x-1 nella forma $x-1 = 2^s.d$, con d dispari.*
- *a = RANDOM(1,2,…,x-1)*
- *if $<a^d>_p \neq 1$ e $<a^d>_p \neq -1$ e…e $<a^{2\wedge(s-1).d}>_p \neq -1$ then return("composto")*
  else return("primo")

In base alla discussione precedente, se l'algoritmo restituisce "composto" allora effettivamente l'ingresso x è un numero composto. Potrebbe però succedere che l'ingresso x è composto, ma viene estratto a tale che $<a^d>_p = 1$ o $<a^d>_p = -1$ o $<a^{2.d}>_p = -1$ o … o $a^{2\wedge(s-1).d}>_p = -1$. In questo caso l'algoritmo darebbe erroneamente la risposta "primo"; si può però dimostrare che questo accade per al più 1/4 dei possibili valori di a, e quindi si presenta la seguente situazione:

- x è primo. L'algoritmo in ogni caso risponde "primo"
- x è composto. L'algoritmo risponde "primo" con una probabilità al più 1/4.

Il tempo di calcolo dell'algoritmo MILLER-RABIN su ingresso $x = x_1,.., x_n$ è un polinomio in n se abbiamo l'avvertenza di utilizzare per il calcolo di $<a^b>_c$ l'algoritmo di "quadrature successive":

Procedura MODPOT(a, b, c)
if b=1 then return $< a >_c$
else if (b pari) then $z = MODPOT(a, \frac{b}{2}, c)$
return $< z^2 >_c$
else $z = MODPOT(a, b\text{-}1, c)$
return $< a \cdot z >_c$

Iterando t volte in modo indipendente l'algoritmo di MILLER-RABIN, si riesce a ridurre la probabilità di errore a $1/4^t$.

Esempio 4.3: Generatore casuale di primi nell'intervallo [2,x]

Il problema richiede di disegnare un algoritmo randomizzato che, avendo in ingresso l'intero $x = x_1,.., x_n$ in notazione binaria, seleziona a caso con distribuzione uniforme un primo p tale che $2 \le p \le x$. Una soluzione può essere:

Algoritmo RANDOMPRIMO*(un intero* $x = x_1,.., x_n$ *in notazione binaria)*

- *for* $1 \le j \le n$ *do* y_j*=RANDOM({0,1})*
- *let be* $y = y_1,.., y_n$ *in notazione binaria*
- *while* $x<y$ *o y è composto do for* $1 \le j \le n$ *do* z_j*=RANDOM({0,1})*
  $y = z_1,.., z_n$
- *return(y)*

Questo algoritmo richiama una procedura per decidere se un intero è composto: tra le procedure che lavorano in tempo polinomiale segnaliamo la procedura randomizzata di Miller-Rabin, o quella deterministica di AKS.

Chiaramente l'algoritmo dà in uscita un intero y che è primo e che è al più x. Poiché ci sono sono approssimativamente x/ln x numeri primi al più x, la probabilità che un intero z scelto a caso tra 1 e x sia un primo è (x/ln x)/x = 1/ln x: si esce quindi dal ciclo while dopo un numero aspettato di ln x iterazioni. Poiché $x<2^n$, vale che ln x = O(n), quindi l'algoritmo ha un tempo aspettato polinomiale.

5. LA CLASSE RP

Ci limitiamo in questa sezione a considerare problemi di decisione, cioè problemi che per ogni istanza ammettono due possibili risposte: vero, falso. Fissato un problema di decisione, indichiamo con L le istanze che rendono vera la questione.

Richiamiamo che il problema L è risolto in modo non deterministico se c'è il seguente protocollo di comunicazione tra un VERIFICATORE V e un DIMOSTRATORE:

- il VERIFICATORE, avendo x in ingresso, invia al DIMOSTRATORE l'istanza x.
- il DIMOSTRATORE, ricevuta x, seleziona in modo non deterministico una parola w all'interno di un insieme di possibili parole, ed invia w al VERIFICATORE.
- il VERIFICATORE, avendo in ingresso (x,w), applica un algoritmo deterministico di verifica V che ha due possibili risposte: "successo" o "fallimento".

Deve inoltre valere:

- se x è in L, allora c'è almeno una parola w selezionabile dal DIMOSTRATORE tale che V(x,w)= "successo". Le parole per cui V(x,w)= "successo"saranno dette "testimoni"
- Se x non è in L, allora per qualsiasi parola w selezionata dal DIMOSTRATORE vale che V(x,w)="fallimento"

L sta in NP se:

- le parole w prese in considerazione dal DIMOSTRATORE quando l'ingresso è x hanno una dimensione polinomiale nella dimensione |x|: $|w|=O(|x|^h)$ per un h opportuno. Supporremo inoltre, senza perdita di generalità, che le parole che il DIMOSTRATORE può scegliere siano tutte di ugual lunghezza.
- L'algoritmo di verifica V lavora in tempo polinomiale nella dimensione di x.

Se ulteriormente, quando x è in L, almeno metà delle parole che il DIMOSTRATORE può scegliere sono "testimoni" (vale cioè V(x,w)="successo") allora diremo che il linguaggio L sta nella classe RP. Sotto queste ipotesi, l'appartenenza al linguaggio L può essere decisa dal seguente algoritmo randomizzato:

$A_{RANDOM}(x)$
S = RANDOM(parole che il DIMOSTRATORE può scegliere su ingresso x)
Return(V(x,S))

Osserviamo ora che, se x non sta in L, allora l'algoritmo randomizzato risponde correttamente "fallimento". Viceversa, se x sta in L, può succedere che l'algoritmo risponda "successo", commettendo errore; la probabilità è tuttavia al più 1/2.

Questo porta a concludere che, se l'algoritmo risponde "successo", allora la risposta è corretta, se invece risponde "fallimento" allora la probabilità di errore è 1/2.
Questa probabilità di errore può essere diminuita ripetendo t volte l'algoritmo in modo indipendente, e accettando quando si è ottenuto almeno un successo. Indicando con $A_{RANDOM}(x)$ il risultato di una esecuzione di A su ingresso x, si ottiene il seguente algoritmo probabilistico:

Input(x)
For k=1, t do R[k] = $A_{RANDOM}(x)$
If esiste k (1≤k≤t) per cui R[k] = "successo" then "successo"
else "fallimento"

La probabilità di errore è limitata in questo caso da $1/2^t$. Se δ è il livello di rischio che riteniamo accettabile, basta scegliere t tale che $1/2^t = \delta$, cioè:

$$t = \lg 1/\delta$$

Si osservi come la crescita di t sia estrememente contenuta in termini di δ. Ad esempio, basta ripetere l'algoritmo 10 volte per ridurre la probabilità d'errore a 10^{-3}.

Una metodologia molto usata per disegnare algoritmi probabilistici che attestano che un problema è in RP è il "fingerprinting (=impronta), che abbiamo già applicato in Esempio 1 della precedente sezione. Supponiamo di voler decidere se due elementi x,y di grande dimensione sono uguali, avendo a disposizione una famiglia H di funzioni. Si può procedere con l'algoritmo randomizzato:

- *h = RANDOM(H)*
- *If h(x)=h(y) then return "successo" else return "fallimento"*

Se la funzione h è calcolabile efficientemente e se h(x), che è detta "impronta di x", è una "compressione" di x, allora il precedente algoritmo è efficiente. Chiaramente, se l'algoritmo restituisce "fallimento" , e quindi vale che h(x) ≠ h(y), risulterà che x≠y. In caso contrario esiste la possibilità di errore, e la classe H deve essere scelta attentamente in modo tale che la probabilità di errore sia 1/2.

Esempio 5.1: **Non Commutatività di matrici**

Il problema ha come istanza due matrici nxn A, B di interi, e si richiede di verificare se esse non commutano, cioè se A.B = B.A.

L'algoritmo immediato richiede di moltiplicare A.B, B.A e confrontare i risultati. Il compito computazionalmente più costoso è quello relativo al prodotto di matrici: esso può essere eseguito in O(n^3) operazioni di somma e prodotto con l'usuale algoritmo di moltiplicazione righe per colonne; non è invece alcun algoritmo deterministico che realizzi la moltiplicazione di matrici in tempo O(n^2).n Non si potrà comunque scendere sotto n^2 passi, poiché la sola lettura di una matrice nxn richiede n^2 passi,

Osserviamo che un modo di "comprimere" una matrice nxn X è quello di scegliere a caso un vettore a di dimensione 1xn in $\{-1,1\}^n$ e calcolare z = aX: in questo caso il calcolo avviene in O(n^2) passi e la "impronta" di X è il vettore 1xn z. Osserviamo infine che, a causa della proprietà associativa del prodotto di matrici rettangolari, vale:

- a.(A.B) = (a.A).B , a.(B.A) = (a.B).A

Questo dà luogo al seguente algoritmo randomizzato per decidere se A.B ≠ B.A:

Commuta$_{\text{RANDOM}}$*(due matrici nxn A,B)*
- *a = RANDOM(*$\{-1,1\}^n$*)*
- $v_1 = a.A$; $v_2 = v_1.B$; $z_1 = a.B$; $z_2 = z_1.A$;
- *if* $v_2 \neq z_2$ *then "successo" else "fallimento"*

L'algoritmo ha complessità ottimale O(n^2).

Per quanto riguarda la correttezza, osserviamo che se $v_2 \neq z_2$ allora ovviamente AB≠ BA, quindi A e B non commutano. Potrebbe però succedere che $v_2 = z_2$ ma AB≠BA: in tal caso l'algoritmo risponderebbe in modo erroneo. E' quindi necessario stimare la probabilità di errore P_e, che è la probabilità di estrarre a in $\{-1,1\}^n$ tale che aAB=aBA con AB≠BA. Posto C=AB-BA, risulta aC=0 mentre C≠0.

A tal fine osserviamo che almeno una colonna $c = (c_1,.., c_n)^T$ di C non è nulla, e supponiamo per semplicità che sia $c_1 \neq 0$. Osserviamo che, dati $a' = (1, a_2,.., a_n)$ e $a'' = (-1, a_2,.., a_n)$, deve essere $\sum_{k=1,n} a'_k c_k \neq 0$ oppure $\sum_{k=1,n} a''_k c_k \neq 0$. Se infatti fosse $\sum_{k=1,n} a'_k c_k = \sum_{k=1,n} a''_k c_k = 0$, sarebbe $c_1 + \sum_{k=2,n} a_k c_k = -c_1 + \sum_{k=2,n} a_k c_k$, e questo implicherebbe $c_1 = 0$, mentre sappiamo che $c_1 \neq 0$. Questo significa che per almeno la metà dei vertici a in $\{-1,1\}^n$ deve valere che $\sum_{k=1,n} a_k c_k \neq 0$, pertanto la probabilità d'errore P_e è al più ½.

6. LE FUNZIONI "ONE_WAY".

Abbiamo visto che un algoritmo probabilistico calcola la funzione f(x) se, su ingresso x, la probabilità che dia in uscita f(x) è almeno 3/4; abbiamo inoltre visto come sia possibile ridurre drasticamente la probabilità di errore con un modesto aumento del tempo di calcolo, ripetendo l'algoritmo in modo indipendente e scegliendo il risultato che compare con maggior frequenza . Possiamo allora considerare "praticamente risolubile", un qualsiasi problema che possa essere risolto da un algoritmo probabilistico che lavora in tempo polinomiale. Riscriviamo allora la TESI DI CHURCH ESTESA:

- Un problema è "praticamente risolubile" se ammette un algoritmo randomizzato risolutivo che lavora in tempo polinomiale.

La contronominale della precedente affermazione porta a descrivere i problemi per cui non esistono "pratici" algoritmi risolutivi:

- Un problema "non è praticamente risolubile" se non può essere risolto da alcun algoritmo probabilistico che lavora in tempo polinomiale.

La seguente nozione può essere letta come una applicazione di questo punto di vista:

Definizione: una funzione invertibile $f:\Sigma^*\rightarrow\Sigma^*$ è detta *funzione one-way* se:
- f può essere calcolata da un algoritmo che lavora in tempo polinomiale
- la sua inversa f^{-1} non può essere calcolata da alcun algoritmo randomizzato che lavora in tempo polinomiale nel seguente senso: per ogni algoritmo randomizzato Alg che lavora in tempo polinomiale e per ogni polinomio p(n), per n sufficientemente grande e per parole w in Σ^n selezionate a caso, la probabilità che w=Alg(f(w)) è al più 1/p(n).

Questa nozione, che, come si vede, è derivata genuinamente in termini di complessità computazionale, ha trovato applicazioni in varie aree, dalla crittografia al commercio elettronico; il prossimo paragrafo è dedicato ad una breve esposizione di alcune sue applicazioni.
L'esistenza di funzioni one-way è un difficile problema teorico che, come il similare problema P =? NP, rimane aperto. Tuttavia alcune funzioni sono ragionevoli candidati. Una di queste è la moltiplicazione di interi in rappresentazione binaria. Siano infatti p, q numeri primi rappresentati con n bit in notazione binaria:
- M(p,q) = p.q può essere calcolato in tempo n^2.
- L'inversa di M richiede, dato z=M(p,q), di ricostruire p, q fattorizzando z. Poiché ad oggi non esistono algoritmi randomizzati che fattorizzano un intero di n bit in tempo polinomiale in n, M è candidata ad essere una funzione one-way.

7. APPLICAZIONI: CRITTOGRAFIA A CHIAVE PUBBLICA

Poniamo il seguente problema: A vuole inviare a B un messaggio w su un canale cui può accedere C, in modo che C non riesca a individuare il contenuto del messaggio. Situazioni di questo tipo sono abbastanza comuni, e non stupisce che ci sia un "corpus di conoscenze" su problemi simili raggruppati in una disciplina chiamata crittologia. La crittologia è a sua volta divisa in due settori distinti: la crittografia, che studia come progettare crittosistemi, cioè sistemi che risolvono il problema proposto, e la crittoanalisi, che studia invece come "rompere" tali sistemi permettendo a C di accedere al messaggio.

L'idea di base per realizzare un crittosistema è quella di considerare una famiglia di funzioni $f_\alpha:\Sigma^*\rightarrow\Sigma^*$, individuate da un parametro α detto "chiave". Tali funzioni sono invertibili, e sia f_α che la sua inversa f_α^{-1} devono essere calcolabili efficientemente. Se A e B, ma non C, condividono la conoscenza di una chiave α , e quindi di f_α e f_α^{-1}, A può inviare a B un messaggio w come segue:

- *A cifra w con $y=f_\alpha(w)$; w è il messaggio in chiaro e y è il messaggio cifrato. A invia y a B sul canale cui può accedere C.*
- *B riceve y e lo decritta applicando f_α^{-1}, ottenendo $f_\alpha^{-1}(f_\alpha(w))=w$.*

Anche se C può accedere a y, non è in grado di ricostruire w in quanto non conosce f_α^{-1}.

Più formalmente, un crittosistema è una tripla <W,T,S> dove:

- W è un insieme di testi in chiaro. Tipicamente $W=\Sigma^r$; se un messaggio è più lungo di r, allora può essere comunque decomposto il blocchi di lunghezza r.
- T è un insieme di possibili testi cifrati. Tipicamente $T=\Gamma^s$.
- S è l'insieme delle chiavi. Ogni chiave α individua una funzione invertibile $E_\alpha:W\rightarrow T$ con la sua inversa $D_\alpha:T\rightarrow W$.

Le specifiche che un crittosistema deve verificare sono:

- Sia E_α che D_α devono essere calcolabili efficientemente
- Senza una esplicita conoscenza di α, deve essere impossibile, o almeno "computazionalmente difficile", calcolare w noto $E_\alpha(w)$.

Si può osservare che, nel modello di crittosistema precedentemente descritto, si può tranquillamente scambiare il ruolo del ricevitore (B) e trasmettitore (A): per questo motivo questi sistemi sono detti *simmetrici*. Ci sono situazioni in cui questi crittosistemi presentano limiti. Ad esempio, in un sistema in cui B vuol ricevere messaggi da più trasmettitori $A_1,\ldots, A_z$ su un canale cui può accedere C, se si utilizza la stessa chiave basta un traditore per danneggiare tutto il sistema.

Una soluzione a questo problema è basata sul concetto di funzione one-way. Supponiamo infatti che le funzioni f_α siano one-way (cioè tale che f_α sia calcolabile efficientemente, mentre f_α^{-1} sia computazionalmente difficile da calcolare), con la

caratteristica ulteriore che sia invece efficientemente calcolabile sulla base di un “segreto” d che dipende da α. In questo caso:

- *Il ricevitore B, che conosce f_α e d, rende pubblica la chiave* α*, quindi la funzione f_α ; sia $A_1,\ldots, A_z$ che C conoscono quindi la chiave* α*.*
- *Il messaggio w_k viene cifrato in $y_k = f_\alpha (w_k)$ da A_k e spedito a B.*
- *B, che conosce il segreto d, può decrittare il messaggio calcolando $f_\alpha^{-1} (y_k)$ utilizzando il segreto d; C può conoscere $y_k = f_\alpha (w_k)$, ma non è in grado di decrittare in quanto non conosce d.*

Questi tipi di crittosistemi vengono detti *“a chiave pubblica”,* poiché la chiave α è stata resa nota a tutti. La difficoltà di decifrazione da parte di C è legata al fatto che la funzione è one-way, quindi il tempo necessario ad invertire la funzione è più che polinomiale: solo chi conosce il segreto d può calcolarne l’inversa efficientemente. Si osserva inoltre che il trasmettitore non conosce il segreto: questi crittosistemi sono *asimmetrici*.

Presentiamo ora il crittosistema a chiave pubblica RSA. La costruzione di questo crittosistema è motivata dal seguente teorema di teoria dei numeri:

Fatto 7.1: Siano p,q due numeri primi, d sia primo rispetto a (p-1)(q-1) ed e sia tale che $\langle e.d\rangle_{(p-1)(q-1)} = 1$. Posto n=p.q, le funzioni $E_{n,e}(x)=\langle x^e\rangle_n$ e $D_{n,d}(x)=\langle x^d\rangle_n$ sono inversa una dell’altra, cioè $D_{n,d}(E_{n,e}(x)) = E_{n,e}(D_{n,d}(x)) = \langle x^{de}\rangle_n = x$.

Esempio 7.1: Il crittosistema di River, Shamir e Adleman (RSA).

Il ricevitore B genera due numeri primi p,q di (grande) dimensione fissata, per esempio col metodo visto nelle precedenti sezioni. Calcola quindi:

$$n = p.q \quad ; \quad \varphi(n) = (p-1).(q-1)$$

Si osservi che φ(n) è esattamente il numero di numeri primi rispetto a n e minori di n; φ è la cosiddetta *funzione di Eulero*.

Il ricevitore B ulteriormente genera a caso un intero d primo rispetto a φ(n): questo può essere ottenuto generando a caso un intero d fino a che non si verifica che MCD(d, φ(n))=1, dove MCD denota l’operazione di Massimo Divisore Comune, che può essere calcolata efficientemente col metodo di Euclide. Infine, il ricevitore determina un intero e tale che $\langle e.d\rangle_{\varphi(n)} = 1$; questo può essere ottenuto, per esempio, risolvendo l’equazione diofantea $x.d + y.\varphi(n) = 1$ col metodo dei convergenti (Euclide), computazionalmente efficiente.

A questo punto il ricevitore rende pubblica la chiave α =(n,e) e tiene d come segreto. Il trasmettitore A_k può ora cifrare il messaggio w, interpretato come intero minore di n in rappresentazione binaria, applicando la funzione:

$$E_\alpha(w) = \langle w^e\rangle_n$$

Il messaggio cifrato $z = \langle w^e \rangle_n$ può essere decrittato da B, che conosce il segreto d, applicando la funzione:

$$D_\alpha(z) = \langle z^d \rangle_n = w$$

Da Fatto 7.1 sappiamo infatti che, per ogni $w<n$, $D_\alpha(E_\alpha(w))=w$.

Si osservi che, anche se C conosce (e,n) e il messaggio cifrato $z = \langle w^e \rangle_n$, C non può ricostruire w, almeno in tempo randomizzato polinomiale. Per poterlo fare, visto che non conosce il segreto d, dovrebbe fattorizzare n determinando p,q, calcolare in seguito $\varphi(n) = (p-1).(q-1)$ ed infine determinare d risolvendo l'equazione diofantea $x.e + y.\varphi(n) =1$. Il punto critico è la fattorizzazione di n nei due fattori p,q: non è attualmente noto nessun algoritmo polinomiale per tale operazione!

Abbiamo ipotizzato che il messaggio w sia minore di n; in caso contrario, w può essere decomposto in blocchi di ugual lunghezza, che denotano in binario numeri minori di n. Questi numeri possono essere cifrati col metodo prima descritto.

Esempio 7.2: firma digitale

La "firma digitale" è una possibile applicazione dei sistemi a chiave pubblica. Con "firma digitale" si intende un tipo di firma apponibile ai documenti informatici con gli stessi effetti della firma autografa apposta ai documenti cartacei.

Una possibile soluzione è la seguente.

Se A vuole inviare a B una richiesta r firmata, per prima cosa rende pubblica una chiave α di cui è l'unico a conoscere il segreto d. A tal riguardo, conoscendo d calcola $y=D_\alpha(r)$ e invia a B la coppia (r,y); si osservi che la firma y, in questo caso, dipende anche dalla richiesta r.

Poiché la chiave α è pubblica, B può verificare che $r =E_\alpha(y)$; in tal caso è sicuro che la richiesta proviene da A, unico possessore del segreto d e quindi unico a poter calcolare $y=D_\alpha(r)$; allo stesso modo B può convincere un terzo attore C che è A colui che ha firmato la richiesta r con la firma y. Infine, non può succedere che B si inventi di aver ricevuto una richiesta v controfirmata da A; infatti, la conoscenza di (r,y) e della chiave α non permette di costruire $D_\alpha(v)$ senza conoscere il segreto d. A è quindi protetto da eventuali contraffazioni da parte di B.

8. ALGORITMI GENETICI

Gli algoritmi genetici rappresentano un classe di modelli di calcolo, motivati da metafora naturale e basati su istruzioni probabilistiche. Essi hanno ricevuto considerevole attenzione come algoritmi di ricerca, ed hanno applicazione in varie aree, dall'ottimizzazione all'apprendimento automatico, dalla simulazione nella genetica delle popolazioni alle problematiche di cooperazione in sistemi multi-agenti.

Poiché in questo contesto è comune l'uso di termini in analogia a quelli della 'reale' biologia, è conveniente richiamare rapidamente un minimo di terminologia biologica.

Gli organismi viventi sono formati da cellule: ogni cellula contiene lo stesso numero di '*cromosomi*'. Ad esempio, la cellula umana contiene 23 coppie di cromosomi, per un totale di 46 cromosomi. Ogni cromosoma può essere visto come una stringa di 'DNA', ed è concettualmente diviso in blocchi funzionali detti '*geni*': ogni gene è responsabile di un '*tratto*' dell' individuo (per esempio: il colore degli occhi), e i possibili valori del tratto sono chiamati '*alleli*' (per esempio: occhi azzurri, marroni, neri). Il materiale genetico, cioè l'insieme di tutti i cromosomi contenuti nella cellula, è detto '*genoma*'. La riproduzione può avvenire mediante la '*ricombinazione*' (o crossover), cioè lo scambio di geni tra due cromosomi. Variazioni casuali in un cromosoma sono dette '*mutazioni*'. La '*fitness*' di un organismo, infine, è la capacità di un organismo di adattarsi e riprodursi.

Poiché ora vogliamo sperimentare la possibilità di risolvere con semplici algoritmi genetici problemi di ottimizzazione, in questa sezione descriviamo un problema di ottimizzazione come una coppia $< C , f >$, dove C è un insieme di soluzioni ammissibili e $f : C \rightarrow N$ la funzione obbiettivo. Il problema consiste nel ricercare il massimo globale $\mathrm{argmax}_x f(x)$ di f, cioè la configurazione $c \in C$ per cui $f(c) \geq f(x)$, per ogni x; nei casi reali, usualmente C viene descritto implicitamente, in quanto la sua cardinalità è grande. La dimensione di C rende generalmente impossibile risolvere il problema di ottimizzazione per via esaustiva.

Nell'algoritmo genetico, il termine cromosoma si riferisce alla generica soluzione ammissibile. Nel caso più semplice, che verrà ora trattato, la generica soluzione è codificata come una parola binaria w in $\{0,1\}^1$, che chiameremo *individuo*. I geni, in questo caso, sono i singoli bit, gli alleli valgono '0' o '1' e la funzione di fitness corrisponde alla funzione obbiettivo f: $\{0,1\}^1 \rightarrow N$. Una popolazione di individui viene descritta da un vettore P = (w[1] , …, w[N]) di parole, che può contenere ripetizioni.

Gli algoritmi genetici operano su popolazioni di individui che vengono fatte evolvere con regole generalmente probabilistiche.

Lo schema generale di un algoritmo genetico è allora il seguente:

ALGORITMO G

P = popolazione iniziale
while < non Cond. Arresto > do P = popolazione ottenuta aggiornando la popolazione corrente con regole stocastiche $R_1,\ldots,R_m$

Ogni aggiornamento di popolazione in questo processo è detta '*generazione*', e l'esecuzione di un algoritmo genetico prevede tipicamente un ordine di $10^2 - 10^3$ generazioni. Poiché le regole di aggiornamento sono generalmente probabilistiche, diverse esecuzioni potranno portare a risultati diversi: quel che si può fare è quello di stimare qualche comportamento medio (per esempio, dopo quante generazioni si riesce a determinare la soluzione ammissibile migliore).

La tipica istruzione probabilistica che abbiamo precedentemente introdotto è

S = RANDOM (A)

L'esecuzione assegna ad S un valore in A con *distribuzione uniforme*. Negli algoritmi genetici potremo prevedere istruzioni probabilistiche che richiedono di estrarre a caso una alternativa, all'interno di un insieme A, in accordo ad una data distribuzione D di probabilità, che non è necessariamente quella uniforme:

S = RANDOM (A , D)

Per esempio, l'esecuzione di

S = RANDOM $(\{0,1\},\{D(0)=0.3\ \ ,\ \ D(1)=0.7\})$

ha come effetto di porre in *S* il valore '0' con probabilità 0.3 o '1' con probabilità 0.7.

In generale, una regola stocastiche di aggiornamento in un Algoritmo genetico è una combinazione di alcune regole base, proposte in base a metafora naturale. Le principali di esse, che andiamo a esporre, sono:

- Selezione
- Crossover
- Mutazione.

a) SELEZIONE

Selezionare un elemento da una popolazione $(x[1],...,x[N])$ significa scegliere a caso, tra $x[1],...,x[N]$, l'elemento $x[k]$ con probabilità $\frac{f(x[k])}{\sum_{j=1}^{N} f(x[j])}$: elementi con valori maggiori di fitness hanno più probabilità di essere selezionati; qui il riferimento darwiniano alla fitness come capacità di sopravvivere è immediato.

La selezione di una popolazione di M elementi da una di N elementi può allora essere descritta:

Procedura SELECT ($(x[1],...,x[N]);M$)

$$\textit{for k=1,M do y[k]} = \textit{RANDOM}\left(\{x[j] \mid j=1,N\}, \left\{D(x[j]) = \frac{f(x[j])}{\sum_{i=1}^{N} f(x[i])}\right\}\right)$$

$$\textit{output}(y[1],...,y[M])$$

Va osservato che l'applicazione della selezione a una popolazione può modificare la popolazione privilegiando individui con alta fitness; un limite dell'operazione di selezione è tuttavia l'incapacità di introdurre *nuovi individui* nella popolazione.

b) CROSSOVER (RICOMBINAZIONE)

Dati due cromosomi $x_1...x_l$ e $y_1...y_l$, la loro ricombinazione (crossover) è la coppia di stringhe $x_1'...x_l'$, $y_1'...y_l'$ ottenute selezionando a caso una posizione k $(1 \le k \le l)$ e giustapponendo il k° prefisso di una al k° suffisso dell'altra:

$$x_1'...x_l' = x_1...x_k y_{k+1}...y_l$$
$$y_1'...y_l' = y_1...y_k x_{k+1}...x_l$$

L'operazione di cross-over è chiaramente riferibile al meccanismo di riproduzione, capace di creare bio-diversità introducendo nuovi individui nella popolazione.

Questa operazione può essere descritta dalla seguente procedura:

Procedura CROSSOVER $(x_1...x_l, y_1...y_l)$

$k = \textit{RANDOM}\ (\{1,2,...,l\})$

return $(x_1...x_k y_{k+1}...y_l \ ; \ y_1...y_k x_{k+1}...x_l)$

c) MUTAZIONE

La mutazione di un cromosoma $x_1 \dots x_l$ è il cromosoma $x_1^{'} \dots x_l^{'}$ ottenuto modificando ogni posizione x_k con una preassegnata probabilità P_{mut}, in modo indipendente per ogni k. Nella seguente procedura, con $\bar{x}_K$ intendiamo l'operazione *not* applicata a x_K :

Procedura MUTAZIONE $(x_1 \dots x_l, P_{mut})$

for k = 1,l do $x_K^{'} = RANDOM(\{x_K, \bar{x}_K\}, D(x_k) = 1 - P_{mut}, D(\bar{x}_K) = P_{mut})$

return $(x_1^{'} \dots x_l^{'})$

Osserviamo che anche la mutazione è responsabile della creazione di nuovi individui nella popolazione.

Un semplice algoritmo genetico, che può essere ottenuto combinando le tre regole precedentemente analizzate, è per esempio:

Algoritmo GENETICO (l: lunghezza cromosoma,
N: dimensione popolazione,
P(c): probabilità di CROSSOVER
P(mut): probabilità di mutazione)

P = popolazione iniziale $(x[1], \dots, x[N])$ *(ottenuta con qualche criterio)*

$$\text{while } \neg \text{cond. Stop do} \begin{cases} \text{for } K = 1, \frac{N}{2} \text{ do} \begin{cases} (x, y) = SELECT(P, 2) \\ C = RANDOM(\{0,1\}, \{D(1) = P_C, D(0) = 1 - P_C\}) \\ \text{if } C = 1 \text{ Then } (x, y) = CROSSOVER(x, y) \\ y[2k-1] = x \\ y[2k] = y \end{cases} \\ \text{for } K = 1, N \text{ do } x[K] = MUTAZIONE(y[K], P_{mut}) \end{cases}$$

L'evoluzione dell'algoritmo genetico esplora, in maniera probabilistica, lo spazio delle configurazioni, passando da una popolazione all'altra per un certo numero di generazioni, nella speranza di selezionare individui con alto valore di fitness. Da questo punto di vista, il modello evolutivo può essere visto come una euristica per risolvere un problema di ottimizzazione globale: questo atteggiamento sarà discusso in seguito, dopo aver premesso una tecnica elementare per analizzare alcune caratteristiche della dinamica evolutiva.

8. ANALISI ELEMENTARE DI ALGORITMI GENETICI

Anche se un algoritmo genetico è descritto da un semplice programma probabilistico, l'analisi teorica del suo comportamento è molto complessa, essendo il processo descritto da una catena di Markov in cui gli stati sono le popolazioni e le probabilità di transizione sono calcolabili a partire dalle regole stocastiche di aggiornamento della popolazione. Discutiamo qui un approccio elementare, proposto da Holland nel 1975.

Consideriamo preliminarmente il seguente esempio.

Esempio 8.1:
E' data al tempo t una popolazione di n simboli (con ripetizione) scelti in $\{0,1\}$, ed è fissata una funzione di fitness $f:\{0,1\}\to N^{+}$. Supponiamo che la popolazione al tempo t+1 sia ottenuta applicando il semplice operatore di *SELEZIONE* alla popolazione al tempo t e ottenendo una nuova popolazione di n elementi. Sapendo che la popolazione al tempo t contiene m(t) volte 1, qual è il valore atteso E[m(t+1)] del numero di 1 al tempo t+1?

La soluzione è semplice. La probabilità di scegliere '1' al tempo 't+1' è:

$$\frac{f(1)\cdot m(t)}{f(1)\cdot m(t)+f(0)\cdot(n-m(t))}$$

Ricordiamo ora che il numero aspettato $E[N°_1]$ di 1 in una popolazione di N elementi, in cui la probabilità che un elemento sia 1 è p, è dato da:

$$E[N°_1] = N \cdot p$$

Allora, indicando con $E[m(t+1)]$ il numero aspettato di 1 al tempo t+1, è:

$$E[m(t+1)] = n\cdot\frac{f(1)\cdot m(t)}{f(1)\cdot m(t)+f(0)\cdot(n-m(t))}$$

Osservando che $\frac{f(1)\cdot m(t)+f(0)\cdot(n-m(t))}{n}$ è la fitness media al tempo t, che chiamiamo $\bar{f}(t)$, e supponendo che il numero effettivo $m(t+1)$ di 1 al tempo t+1 sia, almeno approssimativamente, uguale al numero aspettato, concludiamo:

$$m(t+1) \approx E[m(t+1)] = \frac{f(1)}{\bar{f}(t)}\cdot m(t)$$

Possiamo osservare:

1. Se $f(1) >> \bar{f}(t)$, allora il numero di '1' al tempo 't+1' sarà verosimilmente aumentato con un buon fattore moltiplicativo.
2. Se $f(1) << \bar{f}(t)$, allora il numero di '1' al tempo 't+1' sarà verosimilmente abbattuto.

Supponiamo ora che $f(1) > f(0)$: risulterà che $f(1) \geq \bar{f}(t)$, quindi la dinamica della popolazione tenderà a aumentare gli individui 1, che hanno la fitness maggiore.

La teoria di Holland si riferisce all'analisi approssimata di semplici algoritmi genetici, con popolazioni di n individui $(w_1,..,w_n)$, dove ogni individuo è una parola di l bit $x_1..x_l \in \{0,1\}^l$; la fitness è una funzione $f:\{0,1\}^l \to N^+$.

Viene chiamato *Schema* l'insieme di parole in cui sono fissati gli alleli di alcuni geni. Lo schema viene descritto da una parola di lunghezza '*l*' su $\{0,1,*\}$, dove il simbolo '*' sta per '0' o '1'. Ad esempio, lo schema 0*1* individua le parole di quattro bit che hanno '0' al primo posto e '1' al terzo posto:

0*1* = {0010, 0011, 0110, 0111}

Uno schema H è allora un modo intensivo per intendere una particolare proprietà delle parole, e l'analisi di Holland dà l'idea di come si evolve la presenza di individui con quella proprietà da una generazione all'altra.

Fissiamo ora uno schema H. Sia data una popolazione $(w_1,..,w_n)$ al tempo t e sia m(H,t) il numero di parole nello schema H presenti nella popolazione. Vogliamo determinare il numero aspettato E[m(H,t+1)] di parole di H nella popolazione al tempo t+1, se si è applicata una operazione di *Selezione* (Caso 1) o *Mutazione* (Caso 2) (per semplicità, non trattiamo qui l'operazione di *Crossover*).

Caso 1: *Selezione*

La probabilità che venga selezionata una parola di H è:

$$\frac{\sum_{w\in H} f(w_j)}{\sum_j f(w_j)} = \frac{m(H,t).\sum_{w\in H} f(w_j)/m(H,t)}{n.\sum_j f(w_j)/n} = \frac{\bar{f}(H,t)}{\bar{f}(t)} \cdot m(H,t)/n$$

dove $\bar{f}(t) = \frac{\sum_j f(w_j)}{n}$ è la fitness media della popolazione

$\bar{f}(H,t) = \frac{\sum_{w_j\in H} f(w_j)}{m(H,t)}$ è la fitness media di individui della popolazione che stanno nello schema.

Segue allora, moltiplicando per il numero n di individui della popolazione:

$$(1) \qquad m(H,t+1) \approx E[m(H,t+1)] = \frac{\bar{f}(H,t)}{\bar{f}(t)} \cdot m(H,t)$$

Come nell'esempio 1, possiamo osservare:

1. Se $\bar{f}(H,t) >> \bar{f}(t)$ allora il numero di parole dello schema H presenti nella popolazione al tempo t+1 sarà verosimilmente aumentato di un fattore moltiplicativo.

2. Se $\bar{f}(H,t) << \bar{f}(t)$ allora il numero di parole dello schema H presente nella popolazione al tempo t+1 sarà verosimilmente abbattuto.

Quindi la selezione tende a far crescere la popolazione in schemi che hanno una fitness media più alta di quella della media della popolazione; limite della Selezione è tuttavia la sua incapacità di creare nuovi indvidui.

Caso 2: ***Mutazione***

Dato uno schema $H \in \{0,1,*\}^+$, sia w una parola dello schema. Applicando la *Mutazione* a w, lo schema viene distrutto se e solo se si modificano i simboli che nello schema corrispondono a 0 o 1 (non *!): in ogni altro caso, la nuova parola resterà ancora nello schema.

Chiamiamo *ordine dello schema H* il numero ord(H) di simboli 0 o 1 contenuti nello schema. Ad esempio, ord(**0*1*)=2.

Indicando con P_{MUT} la probabilità di mutazione, segue che la probabilità che lo schema H non venga distrutto dalla Mutazione è $(1-P_{MUT})^{ord(H)}$. Ne segue allora che:

$$(2) \quad E[m(H,t+1)] \geq (1-P_{MUT})^{ord(H)} \cdot m(H,t)$$

Il simbolo $\geq$ nella relazione (2) è motivato dal fatto che non abbiamo tenuto conto dei contributi allo schema H provenienti dalla distruzione di altri schema. Mettendo assieme (1), (2) si ottiene (Holland 1975):

Teorema (dello schema) 8.1: se in una popolazione, che al tempo t ha m(H,t) parole di uno schema H, vengono applicate le operazioni *SELEZIONE* e *MUTAZIONE,* il numero m(H,t+1) di parole nello schema H al tempo t+1 È approssimativamente:

$$m(H,t+1) \approx E[m(H,t+1)] \geq \frac{\bar{f}(H,t)}{\bar{f}(t)} \cdot (1-P_{MUT})^{ord(H)} \cdot m(H,t)$$

dove $\bar{f}(t)$ è la fitness media della popolazione, mentre $\bar{f}(H,t)$ è la fitness media delle parole dello schema H nella popolazione.

9. ALGORITMI GENETICI COME EURISTICA PER L'OTTIMIZZAZIONE

Gli algoritmi genetici si presentano come una euristica, motivata in termini di metafora naturale, per risolvere problemi di ricerca o di ottimizzazione globale, anche nel caso di problemi NP-difficili. Con la parola "algoritmo euristico" si intende un algoritmo, basato su qualche idea trasparente (nel nostro caso è data dalla analogia all'evoluzione della specie), che viene proposto come metodo di visita nello spazio di ricerca, nella speranza di trovare una soluzione vicina all'ottima, anche se non c'è nessuna garanzia che ciò accada.

Nonostante questa mancanza di sicurezza di un buon comportamento, varie euristiche sono ampiamente usate, per i seguenti motivi:

- Le euristiche sono usualmente semplici da implementare, richiedendo solo di selezionare alcuni parametri, cosa che può essere fatta con un'analisi sperimentale. Negli algoritmi genetici tali parametri sono, ad esempio, la dimensione della popolazione, il numero di generazioni, il tasso di mutazione.
- Le euristiche si applicano a una vasta classe di problemi, che invece ammettono algoritmi *ad hoc* molto diversificati.

Per poter applicare a un problema un algoritmo genetico, bisogna operare alcune scelte.

- Per prima cosa, è necessario fissare la nozione di *individuo*; nella nostra trattazione, ci siamo limitati a individui descritti da parole binarie. Nel caso generale, individui potranno essere altre strutture dati, quali i vettori, le matrici, i circuiti, i programmi e così via; in questo caso, mentre la regola di selezione di estende immediatamente, una operazione critica risulta essere il crossover che deve essere proposto e disegnato con accuratezza.
- E' poi necessario stabilire una opportuna nozione di *fitness*. Per molti problemi, tale nozione è immediata e l'algoritmo di calcolo della fitness di un dato individuo è trasparente ed efficiente. In altri, e spesso sono quelli in cui l'algoritmo evolutivo ha maggior successo, la nozione di fitness è imprecisa, oppure difficile da calcolare e può solo essere stimata.

Presentiamo in seguito due esempi. Nel primo i concetti di individuo e fitness sono immediati; il secondo, in cui tali concetti sono più oscuri, è un esempio di applicazione che molti considerano un successo del modello evolutivo.

Esempio 9.1: Massimo di una funzione.

Vogliamo determinare il numero intero tra 0 e 10^6 che massimizza la funzione

$$f(x) = x.(10^6-x).$$

Tale problema serve solo a illustrare il metodo, in quanto può essere banalmente risolto da una ricerca locale (il punto di massimo è $10^6/2$).

- L'individuo è descritto dalla rappresentazione binaria di un intero tra 0 e 10^6, quindi è una stringa binaria di 20 bit.

- La fitness dell'individuo che rappresenta l'intero x è calcolabile direttamente da $f(x) = x.(10^6-x)$.

L'esempio seguente si riferisce all'area dei *giochi strategici*, che modellano situazioni in cui vari attori devono prendere decisioni simultaneamente, sapendo che il *guadagno* di ognuno dipende tuttavia anche dalle decisioni degli altri. Alcuni modelli sono stati introdotti con motivazioni economiche ed analizzati da Von Neumann-Morgenstern (1947) ed estesi da Nash (1950).

Esempio 9.2: Strategie per il dilemma del prigioniero

Due prigionieri A e B, colpevoli di un furto, sono interrogati separatamente. Ognuno dei due ha due possibilità di comportamento: "cooperare" col complice (non confessando) o "tradire" il complice (confessando). Se entrambe confessano hanno una pena alta; essi sanno che tuttavia, se l'altro non confessa, la confessione porta a una forte riduzione della pena. I due prigionieri stabiliscono in modo indipendente la loro scelta e la "partita" ha termine con i "guadagni" (inversamente riferiti alle pene) dati dalle seguenti tabelle:

Guadagno di A

	B coopera	B tradisce
A coopera	4	1
A tradisce	6	2

Guadagno di B

	B coopera	B tradisce
A coopera	4	6
A tradisce	1	2

Viene fissato un numero N di partite. Nel confronto tra A e B, ogni "prigioniero" memorizza i risultati delle ultime k partite (in seguito fissiamo ad esempio k=3) e decide la sua mossa in funzione di questi risultati. Si fanno scontrare il prigioniero A e B per N partite consecutive: il guadagno di ogni prigioniero è costituito dalla somma dei suoi guadagni nelle N partite. In generale, si prendono in considerazione M possibili prigionieri, li si fanno scontrare tutti e vince chi ha avuto il guadagno maggiore.

Chiaramente il "dilemma" è costituito da quale strategia il singolo prigioniero dovrà adottare, visto che non è sempre detto che tradire sia vantaggioso o cooperare sia vantaggioso. Con l'obbiettivo di determinare sperimentalmente le strategie migliori, R. Axelrod [R.Axelrod, The Evolution of Cooperation, 1984)] organizzò un torneo, in cui venivano fatti scontrare 63 "prigionieri", che implementavano strategie proposte da vari ricercatori. Curiosamente, il vincitore risultò la strategia più semplice: TIT FOR TAT. TIT FOR TAT propone all'inizio di collaborare e offre collaborazione finché l'altro prigioniero collabora, ma, se tradito, TIT FOR TAT "punisce" l'altro giocatore tradendolo finché non riprende a collaborare.

Il problema lasciato aperto da questo torneo era: esiste una strategia capace di ottenere punteggi migliori di TIT FOR TAT, che si era dimostrata capace di battere strategie proposte da esperti di inferenza statistica?

Chiaramente questo è un problema di ricerca (di una strategia capace di battere TIT FOR TAT). Per analizzare la dimensione dello spazio di ricerca, osserviamo che le possibili decisioni di una partita sono date dalle 4 coppie di bit:

- CC(A confessa, B confessa), TC (A tradisce, B confessa), CT (A confessa, B tradisce), TT (A tradisce, B tradisce).

Poiché la strategia del prigioniero A è descritta dalla legge che associa ai risultati di 3 partite consecutive una scelta tra C(confessa) e T(tradisce), essa è descritta dalla funzione booleana

$$Z = D(X_1Y_1, X_2Y_2, X_3Y_3)$$

in 6 variabili booleane. Il vettore delle uscite di una tale funzione è una parola binaria $z_0,\ldots, z_{63}$ in $\{C,T\}^{64}$ dove

$$z_i = D(x_1y_1, x_2y_2, x_3y_3)$$

e i è l'intero la cui rappresentazione binaria è $x_1y_1x_2y_2x_3y_3$ (si ponga C=0 e T=1).
Ad esempio, z_{35}=D(CT,CC,TT) poiché la rappresentazione binaria con 6 cifre di 35 è 010011, che diventa CTCCTT considerando 0=C, 1=T.

Poiché il numero di tali vettori è circa $1{,}6.\ 10^{19}$, tecniche esaustive non possono essere utilizzate. Una ipotesi possibile è quella di usare algoritmi genetici per "selezionare" strategie buone, sperando che qualcuna di queste possa battere TIT FOR TAT. A tal riguardo:

- Il singolo cromosoma W è una data strategia, individuata dalla parola binaria in $\{C,T\}^{64}$ che identifica le uscite della funzione booleana che descrive la strategia.
- La funzione di fitness f(W) è calcolata fissando un insieme "rappresentativo" di strategie $\{S_1, \ldots, S_M\}$, facendo combattere W con tutte queste strategie e ponendo f(w) uguale alla somma dei guadagni.

Una analisi di sensitività mostrava che, delle 62 strategie partecipanti al torneo (esclusa solo TIT FOR TAT), bastava sceglierne 8 per creare un opportuno insieme rappresentativo. Definita così la fitness, applicando un algoritmo genetico a una popolazione iniziale di 20 strategie per 50 generazioni, si sono ottenuti risultati decisamente interessanti. La maggior parte delle strategie selezionate sono molto simili al TIT FOR TAT ed addirittura alcune risultano migliori.

Come commento finale, va segnalato che l'idea per cui la piena cooperazione può mantenersi come equilibrio in giochi ripetuti, segnalata nei lavori di Axelrod, era ben

nota da oltre vent'anni. Il contributo di Axelrod va invece visto nell'aver segnalato il modello evolutivo come meccanismo per selezionare strategie che implementano tale idea.

10. LA RICERCA LOCALE

Consideriamo un problema di ottimizzazione descritto dalla coppia $< C, f >$, dove C è un insieme di soluzioni ammissibili e $f : C \rightarrow N$ la funzione obbiettivo, di cui vogliamo determinare il minimo globale $\mathrm{argmin}_x\, f(x)$, cioè la configurazione $c \in C$ per cui $f(c) \leq f(x)$, per ogni x.

Se C è di grande cardinalità, è impossibile risolvere il problema con una ricerca esaustiva. C'è quindi un evidente interesse allo sviluppo di algoritmi che forniscano soluzioni anche "non ottime" utilizzando risorse di calcolo ragionevoli. Un metodo generale che si applica ad una grande varietà di problemi è la ricerca locale. Per risolvere il problema $< C, f >$ con ricerca locale, è necessario munire ogni soluzioni ammissibile $c \in C$ di un *intorno* $In(c) \subseteq C$. La struttura del problema di ottimizzazione diventa allora $< C, f, In >$ e questo permette di introdurre in modo del tutto naturale il concetto di minimo locale:

- Minimo locale per $< C, f, In >$ è un elemento $m \in C$ tale che $f(m) \leq f(s)$ per ogni $s \in In(m)$.

Lo schema di algoritmo di ricerca locale, fissato il problema $< C, f, In >$:

Schema di ricerca locale (C, f ,in)
    x := elemento iniziale scelto in C
    while "x non è minimo locale" do x:= elemento $y \in In(x)$ *tale che f(y)<f(x)*
Uscita x

Esempio 10.1: Ottimizzazione quadratica in {0,1}

Data una matrice simmetrica nxn $A = (A_{ik})$, in cui per semplicità considereremo che $A_{kk}=0$ per ogni k, si vuol determinare il vettore booleano $(a_1, .., a_n)$ in $\{0,1\}^n$ che minimizza $Q(x_1, .., x_n) = \sum_{i,k} A_{ik}\, x_i\, x_k$ .Alcuni problemi di ottimizzazione (tra cui Max CUT, Max 2-SAT, Min VERTEX COVER) possono essere descritti in questo modo, con opportune matrici.

Una semplice tecnica di ottimizzazione locale consiste nel considerare $\{0,1\}^n$ come i vertici di un ipercubo: il vertice $(x_1, .., x_n)$ ha come "intorno" gli n vertici che si ottengono da $(x_1, .., x_n)$ modificando il bit in posizione k ($1 \leq k \leq n$), cioè, ponendo $x'=1$ se $x=0$ e $x'=0$ se $x=1$:

$$\mathrm{In}(x_1, .., x_n) = \{(x_1, .., x'_k, .., x_n) \mid 1 \leq k \leq n\}$$

L'algoritmo di ricerca locale diventa:

Algoritmo RicercaLocale *(una matrice nxn A)*

- x *= un vettore iniziale a in* $\{0,1\}^n$
- *while"esiste k per cui* $Q(x_1, ., x'_k, . , x_n)<Q(x_1, ., x_k, . , x_n)$*" do* $x_k = x'_k$
- *return*$(x_1, .. , x_n)$

Osserviamo ora che $Q(x_1, ., x'_k, . , x_n)-Q(x_1, ., x_k, . , x_n) = 2(x'_k - x_k). \sum_i A_{ik} x_i$.
Se ora $Q(x_1, ., x'_k, . , x_n)<Q(x_1, ., x_k, . , x_n)$, $x_k = 0$ implica che $\sum_i A_{ik} x_i <0$, e $x_k = 1$ implica che $\sum_i A_{ik} x_i >0$. In ogni caso, considerando la funzione di Heaviside u(x) definita da u(x)=1 se x> 0 e u(x)=0 se x<0, da $Q(x_1, ., x'_k, . , x_n)<Q(x_1, ., x_k, . , x_n)$ segue che $x'_j = u(-\sum_k A_{jk} x_k)$.

L'algoritmo RicercaLocale può essere allora riscritto:

Algoritmo Rete di Hopfield *(una matrice nxn A)*

- x *= un vettore iniziale a in* $\{0,1\}^n$
- *while"esiste k per cui* $x'_k \neq u(-\sum_i A_{ik} x_i)$*" do* $x_k = u(-\sum_i A_{ik} x_i)$
- *return*$(x_1, .. , x_n)$

Il nome Hopfield è legato al fatto che tale algoritmo simula la dinamica di una rete neurale simmetrica con evoluzione asincrona (rete di Hopfield).

Il numero di passi della ricerca locale su $< C , f, In >$ è al più F.H, dove $F=\text{Max}_x |f(x)|$ e $H= \text{Max}_x |In(x)|$. Se quindi l'intorno di ogni punto non ha un numero eccessivamente alto di elementi e la funzione obbiettivo f non è troppo grande, la ricerca locale è efficiente, anche per grandi valori di $|C|$. Le difficoltà d'altro lato sono:

- L'algoritmo termina in un minimo locale e generalmente non è noto quanto il costo di tale minimo differisce dal costo del minimo assoluto;
- Il minimo che si ottiene dipende dalla soluzione iniziale.

Esistono varie tecniche che cercano di superare queste difficoltà. Per esempio si può ripetere il processo di ricerca più volte, a partire da diverse configurazioni iniziali scelte a caso, scegliendo il migliore tra i vari risultati ottenuti.

Le prossime sezioni sono dedicate ad analizzare una diversa soluzione, ottenuta creando una opportuna visita randomizzata dello spazio delle configurazioni nella speranza di arrivare con buona probabilità nel massimo globale.

11. RICERCA RANDOMIZZATA: CATENE DI MARKOV.

Poiché la ricerca randomizzata può essere modellata da particolari processi stocastici, richiamiamo brevemente, per prima cosa, alcune nozioni sulle catene di Markov.

Consideriamo un insieme finito di stati S = $\{1,2,..,n\}$. Data una sequenza di stati $<n_0,.., n_k,..>$, denoteremo S(t)= n_t lo stato al tempo t. Supponiamo ora di considerare sequenze casuali, in cui la probabilità che lo stato al tempo t+1 sia j dipende solo dallo stato al tempo t (e non da tempi precedenti): tali sequenze saranno dette *catene di Markov*. Tali catene saranno dunque individuate da:

- La distribuzione di probabilità iniziale π_i = Prob $\{S(0) = i\}$ $(1 \le i \le n)$.
- Le probabilità condizionate $M_{sk}(t+1)$ = Prob $\{S(t+1) = k \mid S(t) = s\}$

Una catena è quindi individuata dal vettore 1xn π, che rappresenta la distribuzione di probabilità degli stati al tempo 0, e dalle matrici nxn M(t), che esprimono la probabilità di ogni transizione di stati al tempo t, per ogni t>0. Il vettore π è un *vettore stocastico*, nel senso che $\sum_i \pi_i = 1$ e $\pi_i \ge 0$, e le matrici M(t) sono *matrici stocastiche*, nel senso che, per ogni k, $\sum_i M_{ki}(t)=1$ e $M_{ki}(t) \ge 0$.

Indichiamo con $\pi_i(t)$ la probabilità Prob $\{S(t) = i\}$ $(1 \le i \le n)$ e denotiamo $\pi(t)$ il corrispondente vettore stocastico; $\pi(t)$ denota quindi distribuzione di probabilità degli stati al tempo t. Non è difficile provare che, in notazione vettoriale:

$$\pi(t) = \pi M(1)M(2)....M(t)$$

Una catena di Markov si dice *omogenea* se $M(1)=M(2)=..=M(t)=...=M$, cioè se le probabilità Prob $\{S(t+1) = k \mid S(t) = s\}$ non dipendono da t. In questo caso la distribuzione di probabilità $\pi(t)$ degli stati al tempo t è data da:

$$\pi(t) = \pi M^t$$

Una catena di Markov omogenea M si dice *regolare* (o ergodica) se esiste un tempo z per cui M^z è una matrice con tutte le componenti maggiori di 0: questo significa che la probabilità di transire in z passi dallo stato k allo stato s è positiva, per ogni k,s.

Per le catene regolari vale il seguente importante risultato:
Teorema 11.1: sia M la matrice di transizione associata ad una catena di Markov ergodica. Allora esiste un unico vettore stocastico q le cui componenti q_i sono determinate dalla seguente equazione:

$$q \cdot M = q, \;\; cioè \;\; \sum_{\mathrm{j}} \mathrm{q_j} M_{ji} = q_i \quad per \quad ogni \quad i$$

Inoltre per ogni distribuzione iniziale π vale:

$$q = \lim_{t \to \infty} \pi \cdot M^t$$

La distribuzione limite q non dipende quindi dallo distribuzione di probabilità iniziale; essa è anche detta distribuzione stazionaria della catena.

Il seguente lemma ci dà una condizione sufficiente perché q sia la distribuzione stazionaria di *M*.

Lemma 11.1: sia *M* la matrice di transizione associata ad una catena di ergodica e supponiamo che esista una distribuzione di probabilità q che verifica

(*) $q_i M_{ij} = q_j M_{ji}$ *per ogni i,j.*

Allora q è la distribuzione stazionaria della catena di Markov *M*.
L'equazione (*) è detta *equazíone di bilancio dettagliato*.

Il lemma precedente risulta completamente dimostrato provando che dall'eq.(*) segue che $q \cdot M = q$ e e ricordando che il vettore stocastico q è unico.

Poiché vale $q_i M_{ij} = q_j M_{ji}$ si ha $\sum_j q_i M_{ij} = \sum_j q_j M_{ji} \Rightarrow q_i \sum_j M_{ij} = \sum_j q_j M_{ji}$.

Poiché *M* è una matrice stocastica, si ha $1 = \sum_j M_{ij}$, cioè $q_i = \sum_j q_j M_{ji}$, quindi $q = q.P$

Esempio 11.1:
Riprendiamo il problema "Dilemma del prigioniero". Richiamiamo che le tabelle dei guadagni dei due giocatori A e B sono rispettivamente $\begin{bmatrix} 4 & 1 \\ 6 & 2 \end{bmatrix}$ e $\begin{bmatrix} 4 & 6 \\ 1 & 2 \end{bmatrix}$.

Indichiamo con X_t e Y_t le scelte operate da A e B al passo t e supponiamo che A e B implementino le strategie:

- A implementa TIT FOR TAT, cioè X_0=C, X_{t+1}=Y_t
- B implementa una strategia casuale con Prob[Y_t =C] = *p*, Prob[Y_t =T]=1-*p* , per un assegnato parametro *p*.

Si vuole stimare il guadagno aspettato *g* di A quando A si scontra con B.

La sequenza $X_t Y_t$ di risultati (t≥0) è una catena di Markov omogenea con stati CC,CT,TC,TT tale che:

- Prob {S(t+1) = XC | S(t) = ZX}= *p*
- Prob {S(t+1) = XT | S(t) = ZX}= *1- p*
- Tutte le altre probabilità sono 0

Quindi la matrice stocastica è M = $\begin{bmatrix} p & 1-p & 0 & 0 \\ 0 & 0 & p & 1-p \\ p & 1-p & 0 & 0 \\ 0 & 0 & p & 1-p \end{bmatrix}$ e la distribuzione stazionaria q, ottenuta risolvendo il sistema lineare $q \cdot M = q$, è $q = \left[p^2, p(1-p), \quad p(1-p), (1-p)^2\right]$

Il guadagno aspettato *g* di A è allora *g*=
$4.p^2 + 1.p(1-p) + 6.p(1-p) + 2.(1-p)^2 = -p^2 + 3p + 2$

12. RICERCA RANDOMIZZATA: ALGORITMO DI METROPOLIS.

Utilizzeremo Lemma 1 della precedente sezione è come base per dimostrare la correttezza della seguente costruzione.

Sia *G* una matrice stocastica *simmetrica* che genera una catena di Markov *regolare* e sia q un vettore stocastico $q = (q_1, q_2, \ldots q_n)$ a componenti positive, che identifica una distribuzione non uniforme (cioè $\forall i \;\; q_i > 0 \;\; e \;\; \exists \text{k} \;\; tale\, che \;\; q_k \neq \frac{1}{n}$) .

Dati *G* e q, si può costruire una nuova matrice *M* con la seguente regola:

$$M_{ir} = \begin{cases} G_{ij} & se \;\; i \neq j \;\; e \;\; q_i \leq q_j \\ G_{ij} \cdot \dfrac{q_j}{q_i} & se \;\; i \neq j \;\; e \;\; q_i > q_j \\ 1 - \sum\limits_{\substack{r=1 \\ r \neq i}}^{n} M_{ir} & se \;\; i = j \end{cases}$$

Si verifica immediatamente che *M* è una matrice stocastica e che essa genera una catena regolare; inoltre date le componenti $q_i \;\; e \;\; q_j (i, j = 1,..n)$, analizzando separatamente i tre casi $q_i = q_j, \;\; q_i < q_j, \;\; q_i > q_j$, si ha che la relazione di bilancio dettagliato $q_i \cdot M_{ij} = q_j \cdot M_{ji}$ è sempre verificata. Per Lemma 1 q è allora la distribuzione stazionaria di *M*, ed in particolare $q = q \cdot M$.

Possiamo allora concludere:

Fatto 12.1: Data una qualsiasi matrice *G* stocastica simmetrica che genera una catena di Markov regolare e un vettore stocastico q, tale che $q_i > 0 \;\; (i = 1, \ldots n)$ e esiste k tale che $q_k \neq \frac{1}{n}$, la matrice stocastica *M* costruita con la regola sopra descritta *(costruzione di Metropolis)* genera una catena di Markov che ha come distribuzione limite proprio il vettore q.

Questa costruzione che, sotto opportune ipotesi, associa ad una matrice stocastica simmetrica *G* e ad un vettore stocastico q una nuova matrice stocastica *M* che ammette q come distribuzione limite, è la base teorica di un algoritmo randomizzato (algoritmo di Metropolis), che cerca di risolvere un problema di minimizzazione < C , f , in > cercando di superare il limiti della ricerca locale.

Tale algoritmo è stato introdotto nel 1953 da Metropolis & al., con l'obbiettivo di simulare l'evoluzione di un sistema fisico in un bagno di calore all'equilibrio termico. L'algoritmo genera una sequenza di stati del sistema nel seguente modo: dato lo stato presente *i* del sistema, con energia E_i, lo stato seguente *j* è generato a partire da *i*

applicando un meccanismo stocastico di perturbazione del sistema. La regola stocastica che descrive il meccanismo è la seguente:

- genera a caso un nuovo stato *j* nell'intorno *In(i)* di *i*.
- se E_j è l'energia dello stato s_j, allora:
  se $E_j - E_i \leq 0$ il nuovo stato è *j*
  se $E_j - E_i > 0$ il nuovo stato è *j* con probabilità $\exp(-(E_j - E_i)/T)$

T è un parametro non negativo, che nell'analogia fisica è interpretato come temperatura del bagno di calore. La regola stocastica descritta sopra è detta criterio di *Metropolis* e l'algoritmo che l'accompagna è detto algoritmo di *Metropolis*.

L'equilibrio termico (a una data temperatura *T*) è caratterizzato dalla *distribuzíone di Boltzmann* , rispetto alla quale la probabilità che il sistema sia in uno stato *i* con energia E_i alla temperatura *T* è data da:

$$\Pr ob(i) = \frac{1}{N_0(T)} \cdot \exp\left(-\frac{E_i}{T}\right)$$

(dove $N_0(T) = \sum_j \exp\left(-\frac{E_j}{T}\right)$ è la costante di normalizzazione).
Si osservi che la distribuzione di Boltzman dà probabilità elevate a stati con bassa energia. Possiamo a questo punto ipotizzare un'analogia tra un sistema fisico e un problema di ottimizzazione basata sulle seguenti corrispondenze:

i) Le "soluzioni ammissibili" del problema di ottimizzazione combinatoria corrispondono agli stati del sistema fisico;
ii) Il "costo" di una soluzione è equivalente all'energia del corrispondente stato.

Lo schema d'algoritmo, in cui parametro *c* ha il ruolo di temperatura, è il seguente:

Algoritmo di Metropolis *(C , f , In)*
  x = elemento iniziale scelto in C
  While" Criterio di STOP≠ VERO" do
    y:= Elemento scelto a caso in In(x)
    if f(y)≤ f(x) then x=y
      else x=y con prob $\exp(-(f(x) - f(y))/c)$

Dato *(C , f , In),* consideriamo il grafo < *C* , *E* > dove *(i,j)* ∈ *E* se e solo se *j*∈ *In(i)* .
Se il grafo < *C* , *E* > è connesso (cioè per ogni coppia di nodi esiste almeno una sequenza di lati che li congiunge) e regolare (tutti i nodi hanno lo stesso grado, cioè | *In(x)*| = *g*), allora la seguente matrice $G = (G_{ij})$ è una matrice stocastica simmetrica che genera una catena di Markov irriducibile:

$$G_{ij} = \begin{cases} 0 & se\ j \notin In(i) \\ \dfrac{1}{|In(i)|} & se\ j \in In(i) \end{cases}$$

Data la distribuzione di Boltzmann $q(c)$, con $q_i(c)$ dato da:

$$q_i(c) = \frac{1}{N_0(c)} \cdot \exp\left(-\frac{f(i)}{c}\right)$$

$$\left(N_0(c) = \sum_j \exp\left(-\frac{f(j)}{c}\right)\right)$$

la matrice che si ottiene applicando la costruzione di Metropolis a G e a $q(c)$ è M dove:

$$M'_{ij} = \begin{cases} G_{ij} & se\ i \neq j\ e\ f(j) \leq f(i) \\ G_{ij} \cdot \exp(-(f(j) - f(i))/c) & se\ i \neq j\ e\ \mathrm{f}(\mathrm{j}) > f(i) \\ 1 - \sum\limits_{r \neq i, r \in In(i)} M'_{ir} & se\ i = j \end{cases}$$

L'algoritmo di Metropolis prima descritto genera una realizzazione della catena di Markov associata a *M*. Da Fatto 12.1 possiamo concludere che la distribuzione stazionaria ottenuta è $q(c)$, cioè la distribuzione di Boltzmann, che assegna probabilità alte a soluzioni ammissibili con basso valore della funzione obbiettivo f.

13. RICERCA RANDOMIZZATA: "ANNEALING" SIMULATO

Richiamiamo il principale risultato della scorsa sezione: fissata la temperatura *c*, applicando l'algoritmo di Metropolis per un tempo sufficientemente si approssima la distribuzione stazionaria $q(c)$, che risulta essere la distribuzione di Boltzmann $q_i(c) = \frac{1}{N_0(T)} \cdot \exp\left(-\frac{f(i)}{T}\right)$.

Posto ora $q^* = \lim_{c \to 0} q(c)$ si verifica facilmente:

$$q_i^* = \begin{cases} \frac{1}{|S_{ma}|} & se\, i \in S_{ma} \\ 0 & se\, i \notin S_{ma} \end{cases}$$

dove S_{ma} è l'insieme dei minimi globali di *f*.

q^* è dunque la distribuzione equiprobabile sui minimi assoluti, che assegna probabilità 1 all'evento di trovare un minimo globale. Questo suggerisce che una strategia per determinare con alta probabilità un minimo globale dovrebbe iterare i seguenti passi fino al raggiungimanto della temperatura 0:

1- Si applica l'algoritmo di Metropolis ad una data temperatura in modo di ottenere la distribuzione stazionaria.
2- Si diminuisce la temperatura

L'idea precedente può essere formulata in modo preciso da uno schema di algoritmo, che chiameremo di "annealing simulato" (in italiano: "ricottura simulata"), che andiamo a descrivere. Per prima cosa, fissiamo una funzione *g(k)* con $\lim_{k \to \infty} g(k) = 0$; *g(k)* denoterà la temperatura al passo k e verrà chiamata "strategia di raffreddamento". Lo schema di algoritmo è il seguente:

Algoritmo di Annealing Simulato *(C,F,in)*
X = elemento iniziale scelto in C
For all k>0 do
c = g(k)
y = Elemento scelto a caso in In(x)
if f(y) ≤f(x) then x = y
else x =y con probabilità exp(-(f(x) – f(y))/ c

Questo algoritmo dà luogo a una catena di Markov non omogenea descritta da $\{P'(g(k))\}$, e dipendente dalla cosiddetta "strategia di raffreddamento" *g(k)*.

L'intuizione è che, se si fa tendere “abbastanza lentamente” *g(k)* a *0* per $k \to \infty$, si ottiene una distribuzione stazionaria in cui la probabilità degli stati di energia minima è 1.

Si ha allora a disposizione un algoritmo che minimizza globalmente una funzione costo *f* a patto che la strategia di raffreddamento verifichi opportune condizioni.

Il punto critico dell'algoritmo di annealing simulato è la scelta della funzione *g(k)*. Sappiamo che *g(k)* deve convergere a *0*; tuttavia, se *g(k)* converge “troppo rapidamente”, con alta probabilità l'algoritmo resta “intrappolato” in un minimo locale come succede per la ricerca locale.

Sono state date condizioni su *g(k)* che garantiscono che la catena di Markov non omogenea generata dall'algoritmo converge ad una distribuzione stazionaria rispetto alla quale la probabilità di essere in una soluzione ottima è 1. Una tra le più interessanti (perchè condizione necessaria e sufficiente) è fornita dal teorema di Hajek, che qui ci limiteremo ad enunciare.

Dato il problema $< C, f, In >$, sia *G* il grafo di ricerca locale $G = < C, E >$, dove $(i,j) \in E$ se e solo se $j \in In(i)$. Indichiamo con S_{ml}, S_{ma} rispettivamente l'insieme dei minimi locali e dei minimi assoluti della funzione *f*.
Dati $i, j \in S$, diremo che *j* è *raggiungíbile ad altezza h* da *i* se esiste in *G* un cammino $x_0, x_1, \ldots, x_p$ con $x_1 = i, x_p = j$ tale che $f(x_k) \le h$ *per* $k = 1, \ldots p-1$. Dato un minimo locale non assoluto *i* $(i \in S_{ml} - S_{ma})$, si definisce *profondità* di *i (prf(i))* il più piccolo *h* per cui esiste una soluzione $j \in S$ con $f(j) < f(i)$ che è raggiungibile ad altezza *h* da *i* ; per completezza poniamo che, se $i \in S_{ma}$, allora $prf(i) = \infty$.

Mantenendo l'analogia fisica, la profondità di *i* è in sostanza l'energia minima di attivazione che occorre fornire per portare il sistema dallo stato *i* ad uno stato *j* di più bassa energia. Il seguente risultato chiarisce per quali strategie di raffreddamento l'algoritmo di annealing simulato porta con probabilità 1 a un minimo globale:

Teorema 13.1: sia $g(k)$ una strategia di raffeddamento del tipo $g(k) = \frac{a}{\log(k)}$, per una certa costante a. Allora la convergenza asintotica dell'algoritmo di annealing simulato alla distribuzione equiprobabile sui minimi globali è garantita se e solo se $a \ge prf^*$ dove $prf^* = \max prf(i)$ per $i \in S_{ml} - S_{ma}$.

$g(k) = \frac{prf^*}{\log(k)}$ risulta quindi essere la strategia di raffreddamento ottimale.

Parte B: Algoritmi paralleli.

1. ALGORITMI PARALLELI

In determinate situazioni, la velocità di esecuzione di un compito è l'elemento critico: si pensi alle applicazioni al tempo reale, in cui ad esempio il calcolo della risposta di un filtro deve essere fatto all'interno dell'intervallo di campionamento. In questi casi può essere utile far eseguire il compito a più processori, sperando che l'aumento di risorse sia compensato da una diminuzione del tempo di calcolo. I modelli di calcolo in cui siano presenti più processori che lavorano contemporaneamente sono detti *paralleli*, così come il modello dotato di un solo processore è detto *sequenziale*. In questa sezione considereremo modelli di calcolo paralleli in cui i processori possono essere sincronizzati con un clock globale.

Esempio 1.1: Se un uomo con un secchio svuota una vasca in un tempo t, ci si può aspettare che p uomini svuotino la stessa vasca in un tempo dell'ordine di t/p. La presenza di più esecutori che lavorano contemporaneamente può, in certi casi, ridurre il tempo necessario ad eseguire un compito.

Esempio 1.2: Re Sole soleva farsi vestire dalla servitù. E' possibile sfruttare un arbitrario numero p di addetti per ridurre il tempo di un fattore p?

Esempio 1.3: Si voglia risolvere il problema PROGRAMMAZIONE LINEARE:

Istanza : un vettore a, una matrice A

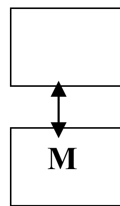
Risposta : Max $\sum_{i=1,n} a_i x_i$ coi vincoli $\sum_{i=1,n} x_i.A_{ik} \geq 0$ (k=1,m)

Questo problema può essere risolto con un algoritmo sequenziale in tempo $O(n^2)$. E' possibile ridurre drasticamente il tempo di esecuzione, per esempio ottenendo tempi $O(\log^2 n)$, sfruttando la presenza di O(n) processori?

I tre problemi precedenti mostrano caratteristiche diverse per quanto riguarda l'esecuzione parallela. Nel primo caso, la presenza di "più processori" può essere sfruttata per ridurre il tempo di esecuzione, nel secondo e terzo caso non si vede come si possano far lavorare i processori in modo che si riesca a ottenere la drastica riduzione di tempo richiesta: apparentemente, il parallelismo è utile per alcuni tipi di problemi, meno per altri. In questo corso non ci occuperemo tuttavia dell'interessante problematica di individuazione dei problemi che possono essere risolti efficientemente con sistemi paralleli.

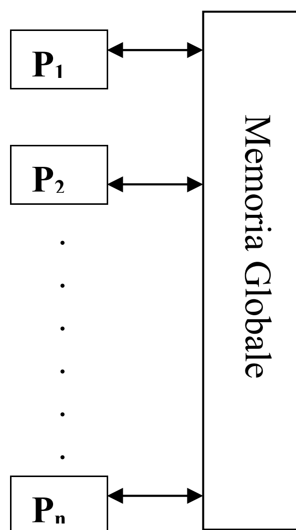
Ci limitiamo qui a introdurre alcuni modelli di calcolo parallelo, in cui i processori possono essere sincronizzati con un clock globale, e a presentare per tali modelli algoritmi risolutivi di semplici problemi.

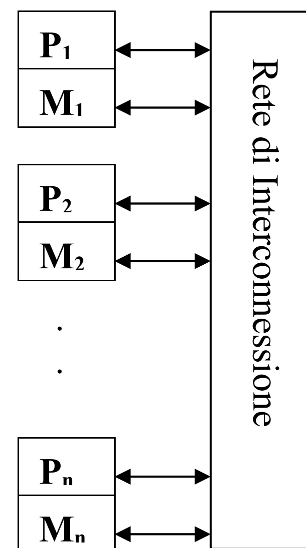
Richiamiamo per prima cosa un classico modello di calcolo sequenziale, cioè la *macchina RAM*; ridotta all'osso, essa consiste di un processore P collegato ad una memoria M attraverso una unità di accesso:



Macchina RAM

Nei modelli di calcolo parallelo, data la presenza di più processori, un elemento critico è la modalità di comunicazione tra processori. I due casi limite sono rappresentati dal modello a *memoria condivisa* e dal modello a *memoria distribuita*.





Memoria condivisa

Memoria distribuita

Nel modello a memoria condivisa tutti i processori possono accedere, attraverso una unità di accesso, alle stesse locazioni di memoria nella stessa unità di tempo. Il meccanismo di comunicazione tra due processori P_k e P_j è molto semplice:

1. P_k scrive il messaggio in un'area di memoria
2. P_j lo legge dalla stessa area

Va segnalato che i processori non sono direttamente connessi e che la comunicazione avviene in tempo costante O(1).
Sfortunatamente, è oggi difficile realizzare tale modello con un adeguato numero di processori; è tuttavia interessante riferirsi a questo modello di calcolo parallelo perché, trascurando i tempi di comunicazione, esso permette all'utente di porre l'attenzione sul potenziale parallelismo disponibile nella ricerca di soluzioni efficienti ad un dato problema.

Nei modelli a memoria distribuita ogni processore può accedere invece solo alla sua memoria privata (locale). La comunicazione avviene qui attraverso l'invio di messaggi attraverso una rete di interconnessione, che può essere descritta da un grafo i cui vertici sono i processori e i lati rappresentano collegamenti diretti tra processori. Poiché in tali reti non tutti i processori sono collegati direttamente, non è possibile ipotizzare tempi di comunicazione costanti, contrariamente a quel che succedeva nel caso a memoria condivisa.

Nelle prossime sezioni presenteremo in qualche dettaglio il modello PRAM (Parallel Random Access Machines), uno dei più popolari modelli a memoria condivisa. In particolare, risolvendo semplici esempi quali SOMMATORIA, SOMME PREFISSE, ORDINAMENTO, sviluppiamo alcune idee per il progetto di algoritmi paralleli; viene infine descritta una tecnica detta "CICLO EULERIANO".

Verranno poi discusse alcune topologie di interconnessione (array lineare, mesh quadrata, albero, ipercubo). Di esse saranno analizzati alcuni parametri critici (grado, diametro, ampiezza di bisezione) e le prestazioni, su problemi *benchmark* quali MASSIMO e ORDINAMENTO, di algoritmi implementati nelle varie topologie.

2. MODELLO P-RAM

Il più semplice modello di calcolo parallelo è quello a memoria condivisa, detto P-RAM, che discutiamo in questo paragrafo. Tale modello consiste di *p-processori* con una memoria globale, condivisa da tutti.

La memoria globale è usata dai processori per scambiarsi dati *in tempo* O(1): perché il processore k e il processore j si scambino un valore, basta che il processore k scriva tale valore in una variabile condivisa e il processore j vi acceda in lettura.

Il calcolo procede per *passi*. Ad ogni passo ogni processore può fare una operazione sui dati che possiede, oppure può leggere o scrivere nella memoria condivisa. In particolare, è possibile selezionare un insieme di processori che eseguono tutti la stessa istruzione (su dati generalmente diversi), mentre gli altri processori restano inattivi; i processori attivi sono sincronizzati, nel senso che eseguono la stessa istruzione simultaneamente e l'istruzione successiva può essere eseguita solo quando tutti hanno terminato l'esecuzione.

Questo modello di macchine è detto di tipo SIMD (Single Instruction Multiple Data). Si possono ulteriormente specificare vari modelli di PRAM, in termini di limitazione agli accessi a memoria condivisa:

1. EREW (Exclusive Read Exclusive Write): non e' ammesso l'accesso contemporaneo da parte di più processori alla stessa locazione di memoria.
2. CREW (Concurrent Read Exclusive Write): l'accesso contemporaneo è permesso in lettura.
3. CRCW (Concurrent Read Concurrent Write): l'accesso contemporaneo è permesso in lettura ed in scrittura.

Per semplicità, in seguito faremo riferimento al modello EREW, il più realistico: in questo caso è compito del programmatore disegnare l'algoritmo in modo che non accadano conflitti in lettura o scrittura. In seguito denoteremo con l'intero i il processore P_i.

La tipica istruzione di un algoritmo parallelo è la seguente:

$$\textit{for all } i\,(a \le i \le b) \textit{ do in parallel} \quad \textit{Operazione}$$

la cui semantica è quella di eseguire *Operazione* su tutti i processori *i* con $a \le i \le b$, mentre gli altri restano inattivi.

Dato un algoritmo parallelo *A*, diremo $T_A(n,p)$ il tempo di esecuzione di *A* su dati di dimensione *n*, quando *A* utilizza p processori; il caso sequenziale si ha ovviamente

quando $p=1$. Obbiettivo di un algoritmo parallelo è quello di diminuire i tempi di calcolo aumentando il numero di processori: si spera allora in un *trade-off* tra il tempo e il costo (numero di processori) che va stimato per valutare l'efficacia di un algoritmo parallelo. Due misure naturali dell'efficacia sono lo *Speed-up* $S_A(p)$ e l'*Efficienza* $E_A(n,p)$:

$$S_A = \frac{T_A(n,1)}{T_A(n,p)}$$

$$E_A = \frac{S_A(p)}{p}$$

Poiché $E_A(n,p) = \frac{T_A(n,1)}{p \cdot T_A(n,p)}$, l'efficienza risulta essere il rapporto tra il tempo dell'algoritmo sequenziale e il tempo totale consumato dai processori, come se fossero usati sequenzialmente.

Dato un algoritmo A che lavora con p processori con una data efficienza E, è in generale possibile estendere l'algoritmo a lavorare con un numero inferiore di processori senza che l'efficienza diminuisca significativamente:

Fatto: se k>1, allora $E_A\left(n, \frac{p}{k}\right) \geq E_A(n,p)$

Dato un algoritmo A che lavora con p processori, basta infatti costruire un algoritmo modificato che utilizza p/k processori. Ad ogni nuovo processore si fa corrispondere un blocco di k vecchi processori: ogni nuovo processore usa al più k passi per emulare il singolo passo parallelo dei k processori corrispondenti. Vale quindi:

$$T_A\left(n, \frac{p}{k}\right) \leq k \cdot T_A(n,p)$$

Osservando che $E_A\left(n, \frac{p}{k}\right) = \frac{T_A(n,1)}{\frac{p}{k} \cdot T_A\left(n, \frac{p}{k}\right)} \geq \frac{T_A(n,1)}{p \cdot T_A(n,p)} = E_A(n,p)$, concludiamo che l'efficienza non diminuisce diminuendo i processori. In particolare, poichè $E_A(n,p) \leq E_A(n,1) = 1$, l'efficienza non può superare 1.

3. ALGORITMI SU P-RAM

In questa sezione presentiamo alcuni elementi di dprogetto di algoritmi paralleli, sviluppando sulla base di esempi algoritmi paralleli per semplici compiti. Trattiamo in particolare due problemi (SOMMATORIA e ORDINAMENTO) e alcune varianti (SOMMA PREFISSA). Presentiamo algoritmi per SOMMATORIA e SOMME PREFISSE che lavorano in tempo O(log n) e un algoritmo, ordinamento bitonico, che ordina un vettore di n elementi in tempo $O(\log^2 n)$. Discutiamo infine una tecnica per il disegno di algoritmi su alberi detta *cammino eluleriano*.

Esempio 3.1: problema SOMMATORIA.

Il problema SOMMATORIA può essere così formulato:

Istanza: interi *a[1], a[2], .., a[n]*

Risposta: $S = \sum_{K=1}^{n} a[k]$

In seguito supponiamo per semplicità che n sia potenza di 2: $n = 2^S$, per qualche s; non c'è perdita di generalità, poiché tra n e 2n c'è sempre una potenza di 2.

Per delineare un primo algoritmo, supponiamo di avere a disposizione $\frac{n}{2}$ processori, detti rispettivamente processore 1, processore 2,.., processore $\frac{n}{2}$. Supponiamo che inizialmente gli interi *a[1], a[2], .., a[n]* siano memorizzati nell'array $(x[1],..,x[n])$. Si può ottenere una soluzione naturale al problema SOMMATORIA procedendo come segue:

al passo 1, ogni processore k $\left(1 \le k \le \frac{n}{2}\right)$ calcola $x[2k] = x[2k] + x[2k-1]$

al passo 2, ogni processore k $\left(1 \le k \le \frac{n}{4}\right)$calcola $x[4k] = x[4k] + x[4k-2]$

. . .

. . .

al passo s, il processore k $(k = 1)$ calcola $x[2^S k] = x[2^S k] + x[2^S k - 2^{S-1}]$

Questa idea è concretizzata dal seguente algoritmo parallelo:

ALGORITMO 1(interi a[1],.., a[n] memorizzati in $(x[1],..,x[n])$*)*

for $j = 1, \log_2 n$ *do*

for all k $\left(1 \le k \le \frac{n}{2^j}\right)$ *do in parallel*

$$x[2^j k] = x[2^j k] + x[2^j k - 2^{j-1}]$$

return $x[n]$

La correttezza dell'algoritmo è provata dimostrando per induzione la verità delle seguenti due affermazioni:

- non vi sono conflitti tra i processori, nè in lettura nè in scrittura.
- al passo j per ogni k $\left(1 \le k \le \frac{n}{2^j}\right)$ la variabile $x[2^j k]$ memorizza il valore $a[2^j k] + ... + a[2^{j-1}(k-1)+1]$

La prova per induzione richiede di provare l'asserzione per j=1 e di provarla per j, supponendola vera per j-1.

Si verifica direttamente che primo passo in $x[2k]$ viene memorizzato $a[2k+1] + a[2k]$. Al passo j viene eseguita l'istruzione $x[2^j k] = x[2^{j-1}.2k] + x[2^{j-1}(2k-1)]$. Supponendo che, per ipotesi di induzione, al passo j-1 $x[2^{j-1}.2k]$ contenga $a[2^{j-1}2k] + ... + a[2^{j-1}(2k-1)+1]$ e $x[2^{j-1}(2k-1)]$ contenga $a[2^{j-1}(2k-1)] + ... + a[2^{j-1}(2k-2)+1]$, concludiamo che al passo j la variabile $x[2^j k]$ contiene $a[2^{j-1}2k] + ... + a[2^{j-1}(2k-2)+1] =$ $a[2^j k] + ... + a[2^j(k-1)+1]$.

Poiché al passo $\log_2$ n la variabile $x[n]$ contiene $a[n] + ... + a[1]$, concludiamo che l'algoritmo risolve il problema in tempo $\log_2 n$ utilizzando n/2 processori:

$$T_{ALG1}\left(n, \frac{n}{2}\right) = \log_2 n$$

Poiché nel caso sequenziale il tempo richiesto è n-1, l'efficienza risulta:

$$E_{ALG1}\left(n, \frac{n}{2}\right) = \frac{n-1}{\frac{n}{2}.\log_2 n} \approx \frac{2}{\log_2 n}$$

L'efficienza di questo algoritmo tende quindi a 0 per n tendente a ∞, sia pur molto lentamente.

Allo scopo di diminuire il numero di processori e nel contempo di migliorare l'efficienza, si osserva che il precedente algoritmo utilizza n/2 processori che lavorano tutti solo nella prima fase del calcolo. Progettiamo allora un secondo

algoritmo (*ALGORITMO 2*) che lavora con p<n/2 processori e che sarà più lento solo nella prima fase. Di questo algoritmo diamo qui una descrizione informale.

ALGORITMO 2

- *Posto* $\Delta = \left\lceil \frac{n}{p} \right\rceil$*, viene assegna al processore k (1≤k≤p) il sottovettore* $(x[\Delta \cdot (k-1)+1]........x[\Delta \cdot k])$*: lavorando in parallelo con gli altri, in tempo* Δ *il processore k calcola sequenzialmente la somma* $S[k]$ *delle componenti di tale vettore in tempo Δ, ponendo il risultato in* $x[k \cdot \Delta]$.
- *Si applica ora ALGORTMO 1 al vettore* $(x[\Delta],..,x[p\Delta])$*, ottenendo la risposta corretta in tempo* $\log_2 p$.

Il tempo complessivamente usato da *ALGORITMO 2* è dato da:

$$T_{ALG2}(n,p) = \frac{n}{p} + \log_2 p$$

La sua efficienza risulta inoltre $E_{ALG2}(n,p) = \frac{n-1}{n + p\log_2 p}$

Scegliendo p in modo tale che $p \cdot \log_2 p = n$, si hanno le due conseguenze:

- $E_{ALG2} = \frac{n-1}{2n} \approx \frac{1}{2}$
- $p \cdot \log_2 p = n$ implica $p \approx \frac{n}{\log_2 n}$

ALGORITMO 2 lavora dunque con $\frac{n}{\log_2 n}$ processori, ha un tempo di calcolo $\log_2 n + \log_2 p \leq 2\log_2 n$ ed una efficienza pari a ½. Il tempo è allora logaritmico $(2\log_2 n)$ con un numero sublineare di processori $\left(\frac{n}{\log_2 n}\right)$.

Esempio 3.2: problema SOMME PREFISSE

Affrontiamo ora una estensione del problema SOMMATORIA, che chiameremo SOMME PREFISSE, così definito:

SOMME PREFISSE:

Istanza: interi *a[1], a[2], .., a[n]*

Risposta: *S[1] = a[1], S[2]=a[1]+a[2], ..., S[n]= a[1]+a[2]+ ...+ a[n]*

La risposta è quindi costituita dai valori $S[k] = a[1] + .. + a[k]$, per ogni *k* da 1 a n. Questo problema può essere considerato un archetipo di vari problemi su liste.

Supponiamo che i numeri *a[1], a[2], .., a[n]* siano inizialmente memorizzati nell'array M=$(M[1], M[2], ..., M[n])$, e utilizziamo una tabella $succ[k]$ che useremo per indirizzare ad M. Inizialmente, la tabella *succ* è data da:

$$succ[k] = \begin{cases} k+1 & k \leq n \\ 0 & \mathrm{k} = n \end{cases}$$

Anche in questo caso, senza perdita di generalità, possiamo ipotizzare che n sia una potenza di 2. Una soluzione parallela per macchine EREW con n processori è la seguente:

ALGORITMO SOMMEPREFISSE(interi $a[1],.., a[n]$ *memorizzati in* $(M[1],..,M[n])$*)*

$$\begin{array}{l} for\ k = 1, \log n \quad do \\ \quad for\ all\ \mathrm{k}(1 \leq \mathrm{k} \leq n) \quad do\ in\ parallel: \\ \qquad if\ succ[k] \neq 0 \quad do \quad \begin{cases} M[succ[k]] = M[k] + M[succ[k]] \\ succ[k] = succ[succ[k]] \end{cases} \\ risposta: (M[1], M[2], ..., M[n]) \end{array}$$

La correttezza dell'algoritmo è provata dalle seguenti considerazioni.

- Si osserva per prima cosa che, dopo *t* passi, la tabella *succ* è tale che:

  $$succ[k] = \begin{cases} k+2^t & k+2^t \leq n \\ 0 & \text{altrimenti} \end{cases}$$

  Inizialmente *succ* descrive quindi una lista di *n* elementi; dopo *t* iterazioni essa descrive 2^t liste di $\frac{n}{2^t}$ elementi e quindi, in particolare, dopo $\log_2 n$ iterazioni descrive *n* liste di 1 elemento.

- Si prova poi, per induzione, che alla iterazione *t* il valore M[k]= è:

  $$M[k] = \begin{cases} a_K + .. + a_{K+1-2\wedge t} & se \quad k > 2^t \\ a_K + .. + a_1 & se \quad k \leq 2^t \end{cases}$$

- Osserviamo infine che, durante l'esecuzione, il vettore *succ* è tale che, se $i \neq j,\ \ succ(i) \neq 0,\ \ succ(j) \neq 0$, allora $succ(i) \neq succ(j)$: non si hanno quindi conflitti in lettura nell'esecuzione dell'algoritmo.

Poiché la complessità in tempo è $O(\log_2 n)$ e il numero di processori è *n*, l'efficienza risulta essere $O\left(\frac{1}{\log_2 n}\right)$.

Esempio 3.3: problema ORDINAMENTO

Il problema ORDINAMENTO può essere così formulato:

Istanza: un vettore $(A[1],\ldots,A[n])$, che qui consideriamo, per semplicità, a componenti intere positive tutte diverse tra loro; supponiamo inoltre che n sia potenza di 2.
Risposta: una permutazione $p(1), .., p(n)$ tale che $A[p(1)]< \ldots <A[p(n)]$.

Ci limitiamo a considerare algoritmi di ordinamento per confronto. E' noto che, nel caso sequenziale, ogni algoritmo di ordinamento richiede almeno Ω(n.log n) confronti ed esistono algoritmi sequenziali che richiedono al più O(n.log n) confronti.

Uno di questi algoritmi è il cosiddetto SortMerge, che è una classica applicazione della tecnica "divide et impera". L'idea essenziale di tale algoritmo è di risolvere banalmente il problema per n=2, mentre per n>2:

- avendo in ingresso $(A[1],\ldots,A[n])$, si divide il vettore in due vettori uguali $A_s=(A[1],\ldots,A[n/2])$ e $A_d= (A[1+n/2],\ldots,A[n])$, che vengono ordinati ricorsivamente.
- I due vettori ordinati vengono utilizzati per costruire l'ordinamento del vettore originale, applicando l'operazione di Merge.

Il punto centrale è che, avendo due vettori già ordinati di dimensione n/2, l'operazione di Merge si effettua efficientemente (in ambito sequenziale) in tempo O(n). Se cerchiamo di parallelizzare il precedente algoritmo in maniera efficiente, vediamo che la prima fase non presenta problemi se, avendo n processori, assegniamo ai primi n/2 il primo vettore, ai secondi n/2 il secondo vettore. La difficoltà sta nel trovare un efficiente algoritmo parallelo per l'operazione di Merge, poiché l'algoritmo sequenziale non appare parallelizzabile.

Discutiamo qui un'idea generale che permette di ottenere algoritmo parallelo efficiente per l'ordinamento. Si procederà con una esposizione ad alto livello, che renda chiara l'idea evitando dettagli in prima approssimazione trascurabili. Per prima cosa, consideriamo le due operazioni di giustapposizione e trasposizione:

- Date due sequenze $A=(A[1],\ldots,A[n])$ e $B=(B[1],\ldots,B[m])$, la loro giustapposizione $A\bullet B$ è la sequenza $A\bullet B=(A[1],..,A[n],B[1],\ldots,B[m])$.
- Data una sequenza $A=(A[1],\ldots,A[n])$, la sua trasposta $A^R =(A[n],\ldots,A[1])$.

Osserviamo che, avendo a disposizione un adeguato numero di processori, tali operazioni su P-RAM si effettuano in tempo parallelo O(1).

Ritornando all'algoritmo SORTMERGE, se i vettori $A_s=(A[1],\ldots,A[n/2])$ e $A_d=(A[1+n/2],\ldots,A[n])$ sono già stati ordinati, basterà calcolare $B= A_s \bullet (A_d)^R$ in tempo O(1) e ordinare B. Il punto chiave consiste nell'osservare che B è una sequenza

bitonica e che le sequenze bitoniche possono essere ordinate con un algoritmo parallelo efficiente.

A tal riguardo, diremo che la sequenza $(A[1],...,A[n])$ è *unimodale* se esiste k tale che $A[1]<..<A[k])>A[k+1]>..>A[n])$ oppure $A[1]>..>A[k])<A[k+1]<...<A[n]),$ mentre $(A[1], ...,A[n])$ è *bitonica* se esiste una sua permutazione circolare che è unimodale, cioè se esiste j per cui la sequenza $(A[j],... A[n], A[1],...,A[j-1])$ è unimodale.

Ad esempio, la sequenza (2,3,7,8,4) è unimodale e quindi bitonica, mentre (7,8,4,2,3) non è unimodale ma bitonica, perché la sua permutazione circolare (2,3,7,8,4) è unimodale; la sequenza (1,6,2,5,3,4) non è invece nemmeno bitonica. Si osservi che, se A_s e A_d sono sequenze crescenti, allora $A_s \bullet (A_d)^R$ è unimodale e quindi bitonica.

Data una sequenza $A=(A[1],...,A[2n])$ di 2n elementi, diremo A_{MAX} e A_{MIN} le sequenze $(A_{MAX}[1],..., A_{MAX}[n])$ e $(A_{MIN}[1],..., A_{MIN}[n])$ di n elementi dove:

- $A_{MAX}[k]) = Max\{A[k], A[n+k]\} \quad (k=1,n)$
- $A_{MIN}[k]) = Min\{A[k], A[n+k]\} \quad (k=1,n).$

Per sequenze bitoniche, si possono verificare le seguenti proprietà:

Fatto 3.1: Se A è bitonica, allora:

1

1. Ogni elemento di A_{MIN} è minore di ogni elemento di A_{MAX}.
2. A_{MIN} e A_{MAX} sono bitoniche.

Queste proprietà suggeriscono una semplice procedura che, avendo in ingresso una sequenza bitonica, dà in uscita la sequenza ordinata in modo crescente:

Procedura BitonicMerge(A: un sequenza bitonica di $n=2^k$ elementi)
If n=2 then return $(A_{MIN}[1], A_{MAX}[1])$
        else 1. A_1= BitonicMerge (A_{MIN})
                2. A_2= BitonicMerge (A_{MAX})
                return $(A_1 \bullet A_2)$

La correttezza è dimostrata immediatamente per induzione ricordando che se A è bitonica A_{MIN} e A_{MAX} sono a loro volta bitoniche, e tutti gli elementi di A_{MIN} sono minori di quelli di A_{MAX}. Per quanto riguarda il tempo di esecuzione parallelo $T_{BM}(n)$ di una implementazione del precedente algoritmo su PRAM con n processori, pur senza entrare nei dettagli implementativi si può osservare che il calcolo di A_{MIN} e A_{MAX} a partire da A può essere effettuato in tempo costante O(1) e i passi 1. e 2. possono essere eseguiti in parallelo. Vale quindi:

$$T_{BM}(n)= T_{BM}(n/2)+O(1)$$

La soluzione della precedente equazione di ricorrenza è $T_{BM}(n) = O(\log n)$: su PRAM con n processori una sequenza bitonica può essere ordinata in tempo $O(\log n)$.

Utilizzando la procedura BitonicMerge si ottiene il seguente algoritmo, che diamo in forma ricorsiva senza entrare in ulteriori dettagli implementativi, che permette di ordinare una qualsiasi sequenza (non solo sequenze bitoniche):

Procedura BitonicSort(A: un sequenza di $n=2^k$ elementi)
If n=2 then return (A_{MIN} [1], A_{MAX} [1])
Else 1. A_1= BitonicSort (A[1],...,A[n/2])
2. A_2= BitonicSort (A[1+n/2],...,A[n])
return BitonicMerge($A_1 \bullet A_2^R$)

La correttezza dell'algoritmo è provata facilmente per induzione, osservando che se A_1e A_2 sono ordinate, allora è $A_1 \bullet A_2^R$ è bitonica e quindi *BitonicMerge*($A_1 \bullet A_2^R$) è ordinata.

Osserviamo che su PRAM con n processori i passi 1. e 2. possono essere eseguiti in parallelo. Detto $T_{BS}(n)$ il tempo necessario a ordinare una sequenza di n elementi con l'algoritmo BitonicSort, si ha quindi:

$T_{BS}(n)= T_{BS}(n/2)+ T_{BM}(n)$

Ricordando che $T_{BM}(n) = O(\log n)$, si ottiene $T_{BS}(n)= T_{BS}(n/2)+ O(\log n)$. Risolvendo quest'ultima equazione si ha che $T_{BS}(n) = O(\log^2 n)$.

In conclusione, su P-RAM con n processori una sequenza di n elementi può essere ordinata in tempo $O(\log^2 n)$ con l'algoritmo *BitonicSort*.
Le idee e i risultati sulle sequenze bitoniche possono essere utilizzati come base per disegnare una classe di algoritmi di ordinamento (per confronto). Ad esempio, la versione sequenziale dell'algoritmo di *BitonicSort* è meno efficiente di *SortMerge*, poiché lavora in tempo $O(n. \log^2 n)$, mentre il *BitonicSort* può essere efficientemente realizzato su su varie architetture parallele, quali le reti di interconnessione come le mesh e l'ipercubo.

4. TECNICA DEL "CICLO EULERIANO"

Presentiamo in questo paragrafo una tecnica di base spesso utile nella progettazione di algoritmi paralleli su P-RAM che lavorano su alberi, perché riduce spesso il problema a un problema che lavora sulla più "trattabile" (dal punto di vista del parallelismo) struttura di lista.

Richiamiamo rapidamente alcune nozioni su grafi. Un ciclo euleriano in un grafo orientato *G* è un ciclo che attraversa ogni arco una e una sola volta. E' facile dimostrare che condizione necessaria e sufficiente perché *G* ammetta un ciclo euleriano è che *G* sia connesso e che in ogni vertice il numero di archi entranti sia uguale al numero di archi uscenti.

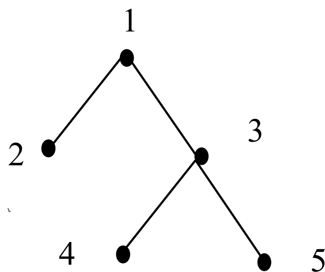
Esempio 4.1: Consideriamo un albero *T*, e sia T_e il grafo orientato ottenuto sostituendo ogni lato $\{i, j\}$ di *T* coi due archi *(i,j)* e *(j,i)* ; poiché in T_e in ogni vertice il numero di archi entranti è uguale al numero di archi uscenti, il grafo T_e ammette un cammino euleriano.

La tecnica del ciclo euleriano consiste nel cercare di risolvere un dato problema su alberi dividendolo nei due passi principali:

(1) Dato un albero *T*, si costruisce in parallelo un cammino euleriano *L* in T_e
(2) Si applica a *L*, visto come lista, un algoritmo parallelo per liste

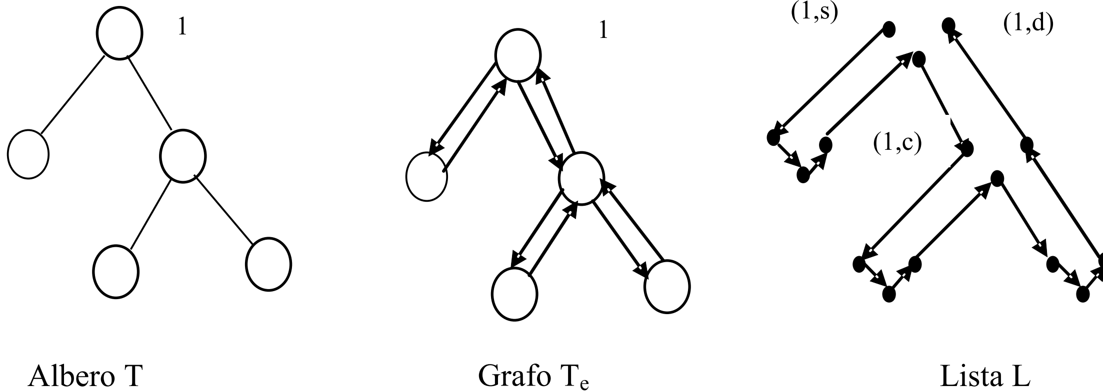
Descriviamo qui un algoritmo parallelo che risolve il problema (1) quando *T* è un albero binario ordinato con radice.

Supponiamo che l'albero binario con radice *T* sia descritto dalle tabelle *sin* (figlio sinistro), *des* (figlio destro), *padre* (padre), come nel seguente esempio.



x	*sin(x)*	*des(x)*	*padre(x)*
1	2	3	0
2	0	0	1
3	4	5	1
4	0	0	3
5	0	0	3

Per ogni vertice *v* dell'albero consideriamo 3 nuovi elementi *(v,s),(v,c),(v,d)*. Un cammino euleriano in T_e può essere descritto dalla lista *L* che ha come indirizzi i nuovi elementi *(v,x)*, e la tabella *succ[(v,x)]* opportunatamente definita, estendendo l'esempio mostrato nella figura seguente:



La tabella *succ* viene costruita quindi nel seguente modo:

se *v foglia*

$$succ[(v,s)]=(v,c)$$

$$succ[(v,c)]=(v,d)$$

$$succ[(v,d)] = \begin{cases} (padre[v],c) & \text{se } v=\sin[padre[v]] \\ (padre[v],d) & \text{se } v=des[padre[v]] \end{cases}$$

se *v nodo interno*

$$succ[(v,s)]=(sin[v],s)$$

$$succ[(v,c)]=(des[v],s)$$

$$succ[(v,d)] \begin{cases} (padre[v],c) & \text{se } v=\sin[padre[v]] \\ (padre[v],d) & \text{se } v=des[padre[v]] \end{cases}$$

Si osservi che, fissato *(v,x),* le operazioni per determinare succ[*(v,x)*] sono locali: avendo a disposizione un processore per ogni elemento *(v,x),* la lista L viene costruita in parallelo in tempo O(1).

Abbiamo visto come, dato un albero T, sia possibile con un algoritmo parallelo efficiente costruire una lista L che individua un cammino euleriano. Mostriamo ora due problemi risolubili con questa tecnica:

- Attraversamento di un albero in preordine.
- Profondità dei nodi di un albero binario.

Entrambe i problemi richiedono di associare opportuni interi ad ogni nodo della lista L e risolvere il problema delle SOMME PREFISSE per la lista.

Esempio 4.2: Attraversamento di un albero in preordine.

Ricordiamo che, per un albero binario T, l'attraversamento in ordine anticipato (preordine) consiste nel visitare prima la radice, nell'attraversare in preordine il sottoalbero di sinistra e poi nell'attraversare in preordine il sottoalbero di destra. La visita in preordine permette in linea di principio di associare ad ogni nodo *v* dell'albero il numero d'ordine *N(v)* di visita del nodo.

Per ottenere un algoritmo parallelo per il calcolo di *N(v)* si consideri il grafo orientato T_e associato a *T* e si pesino con 1 i lati di T_e che "allontanano" dalla radice, con 0 quelli che "avvicinano" alla radice. Fissato un cammino euleriano che parte dalla radice, si consideri un qualsiasi sottocammino che parte dalla radice e arriva a un vertice v: la somma dei pesi degli archi di questo sottocammino è esattamente *N(v).* Questo dà luogo al seguente algoritmo parallelo di attraversamento:

ALGORITMO ATTRAVERSAMENTO (un albero binario *T*)

1. *Dato l'albero T, costruisci la lista L che denota un ciclo euleriano.*
2. *Associa 1 a un vertice (v,x) tale che succ[(v,x)]= (v',x') e v=padre(v'); associa 0 a tutti gli altri vertici.*
3. *Applica alla lista L coi pesi così stabiliti l'algoritmo per SOMMEPREFISSE.*

La correttezza è provata osservando che la somma prefissa fino al vertice (v,s) è il numero d'ordine N(v) del vertice v in una visita di T in preordine.

Poiché il calcolo di L e la definizione dei pesi dei suoi elementi richiede tempo parallelo O(1) e l'esecuzione dell'algoritmo per le somme prefisse richiede tempo parallelo $O(\log_2 n)$, concludiamo che l'attraversamento di un albero in preordine può messere fatto con un algoritmo parallelo in tempo $O(\log_2 n)$.

Esempio 4.3: Profondità dei nodi di un albero binario.

Ricordiamo che la profondità di un nodo *v* in un albero binario T è la lunghezza del cammino che va dalla radice a *v*.

Per ottenere un algoritmo parallelo per il calcolo della profondità, si consideri il grafo orientato T_e e si pesano con *+1* il lati di T_e che "avvicinano" alla radice, con *–1* i lati di T_e che "allontanano" dalla radice: si osservi che in tal modo la somma dei pesi di lati che formano un cammino chiuso è 0. La profondità dei vari vertici può essere ottenuto semplicemente costruendo in parallelo in tempo O(1) la lista L associata al cammino euleriano di T_e , calcolando poi in tempo $O(\log_2 n)$ le somme prefisse.

5. RETI DI INTERCONNESSIONE

Nel modello P-RAM precedentemente analizzato ogni processore può scambiare informazione con ogni altro in tempo O(1) grazie alla condivisione della memoria. Questa velocità di comunicazione si riflette nella velocità con cui molti problemi possono essere risolti su tale modello. D'altro lato, è importante ricordare che il modello P-RAM non è fisicamente realizzabile, con le attuali tecnologie, se non quando il numero di processori è modesto.

Modelli alternativi sono le macchine a memoria distribuita. In questo caso, ogni processore può accedere soltanto ad una sua memoria locale e i processori possono scambiarsi informazioni solo attraverso una opportuna rete di interconnessione; supporremo qui che la sincronizzazione avvenga grazie ad un orologio globale.

Una rete di interconnessione può essere astrattamente descritta da un grafo, i cui nodi sono i processori e i lati sono coppie di processori che possono comunicare direttamente. Per semplicità, supponiamo qui che la comunicazione sia bidirezionale: se A comunica direttamente con B allora anche B comunica direttamente con A. Il grafo che descrive l'interconnessione è allora un grafo non orientato G = (V,E).

Importanti parametri di una rete di interconnessione sono il suo *grado*, il suo *diametro* e la sua *ampiezza di bisezione*.

- *Grado della rete* G = (V,E).
  Il grado di un vertice $i \in V$ è il numero d(i) di lati uscenti da i. Il grado d(G) della rete è il massimo dei gradi dei vertici:
  $$d(G) = Max_i\, d(i)$$
  Reti di grado elevato sono di difficile costruzione: un basso grado risulta allora una caratteristica desiderabile.

- *Diametro di comunicazione* della rete G = (V,E). La distanza tra due vertici $i \in V$ e $j \in V$ è la lunghezza D(i,j) del più breve cammino da i a j. Il diametro di comunicazione D(G) è la massima distanza tra vertici:
  $$D(G) = Max_{i,j}\, D(i,j)$$
  Un basso diametro è una caratteristica desiderabile, poiché permette un trasferimento veloce di informazione tra una qualsiasi coppia di processori.

- .*Ampiezza di bisezione* della rete G = (V,E).La ampiezza di bisezione B(G) del grafo G è il minimo numero di lati che si devono togliere in modo che il nuovo grafo ottenuto sia formato da due componenti disconnesse contenenti (approssimativamente) lo stesso numero di vertici. La ampiezza di bisezione è un limite inferiore alla quantità di informazione che metà dei processori possono scambiare con i rimanenti in un passo di calcolo. Reti con alta

ampiezza di bisezione permettono di smistare tra i processori in poco tempo grandi quantità di dati; per contro, tali reti risultano di difficile realizzazione.

Un ulteriore parametro che permette di valutare le caratteristiche di una rete è il tempo di esecuzione del miglior algoritmo per la soluzione di un dato problema. In questa breve rassegna utilizzeremo due problemi di riferimento per il confronto di reti:

- Determinazione del MASSIMO in un vettore
- ORDINAMENTO.

Questi problemi possiedono infatti caratteristiche diverse. La capacità di risolvere efficientemente la determinazione del MASSIMO, simile al problema SOMMATORIA, risiede infatti nella semplice possibilità di far comunicare rapidamente una qualsiasi coppia di processori. Una soluzione parallela efficiente al problema ORDINAMENTO richiede invece di poter trasportare velocemente grandi masse di dati in un singolo passo. Queste caratteristiche portano alle seguenti stime di limiti inferiori al tempo necessario a risolvere MASSIMO e ORDINAMENTO con algoritmi paralleli implementati su reti di interconnessione.

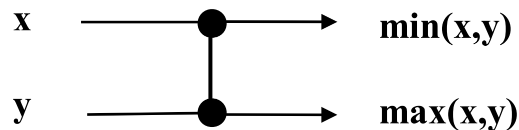
Fatto 5.1: per risolvere il problema MASSIMO di n elementi con una rete di diametro D occorre almeno un tempo D.
Se infatti due processori hanno distanza maggiore di D, sarà impossibile confrontare tra loro gli elementi contenuti in questi processori in D passi.

Fatto 5.2: per risolvere il problema ORDINAMENTO di n elementi con una rete di ampiezza di bisezione B occorre almeno un tempo n/2B.
Supponiamo infatti che ogni processore contenga all'inizio un elemento da ordinare e che alla fine il processore k debba contenere il k° elemento in ordine crescente. Se i primi n/2 processori contengono gli n/2 elementi più grandi, alla fine del processo essi devono contenere gli n/2 elementi più piccoli. Ad ogni passo, possiamo tuttavia trasferire in questi processori al più B elementi dagli altri processori, quindi i passi richiesti sono almeno n/2B.

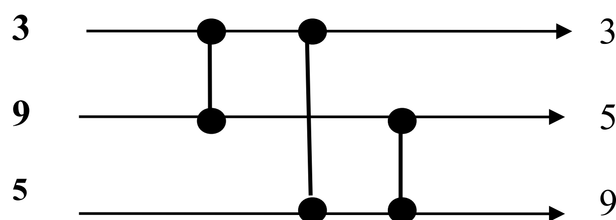
Prima di descrivere algoritmi per l'ordinamento su diverse reti di interconnessione, presentiamo un risultato sulle *reti di ordinamento* (sorting network), che permette di semplificare grandemente le prove di correttezza degli algoritmi.

Molti algoritmi di ordinamento basati su confronti sono "oblivious", cioè le sequenze di confronti per qualsiasi ingresso di data dimensione. In questo caso è semplice descrivere gli algoritmi come circuiti, la cui componente elementare è l'elemento *confronto-scambio*, che riceve in ingresso la coppia di interi (x,y) e dà in uscita la

coppia, ordinata in modo crescente, (min(x,y), max (x,y)):



Una *rete di ordinamento* di n interi può essere graficamente rappresentata da n linee orizzontali, aventi in ingresso gli n interi; ogni operazione di confronto-scambio è rappresentata da una connessione verticale tra due di queste linee. Una rete per l'ordinamento di 3 interi è rappresentata nella figura seguente:



In generale una rete R, avendo in ingresso un vettore di interi $(x_1, \ldots, x_n)$, dà in uscita un vettore di interi $(y_1, \ldots, y_n)$ che denoteremo $R(x_1, \ldots, x_n)$. La rete R è *corretta* se, avendo in ingresso un qualsiasi vettore di interi $(x_1, \ldots, x_n)$, dà in uscita il vettore ordinato $(y_1, \ldots, y_n)$, tale cioè che $y_1 \leq y_2 \leq \ldots \leq y_n$.

Uno strumento che grandemente semplifica le dimostrazioni di correttezza delle procedure di ordinamento su reti è un risultato detto "*principio 0,1*"(Knuth 1972):

Fatto 5.3 *(principio 0,1)*: se una rete di ordinamento R lavora correttamente su vettori con componenti in {0,1}, allora lavora correttamente per qualsiasi vettore di interi.

Per provare questo principio, consideriamo una qualsiasi funzione $f:N\rightarrow N$ monotona, cioè tale che, se $x\leq y$, allora $f(x)\leq f(y)$. Chiaramente $f(\max(x,y))=\max(f(x),f(y))$ e $f(\min(x,y))=\min(f(x),f(y))$; ne segue che, se $(y_1, \ldots, y_n)=R(x_1, \ldots, x_n)$, allora $(f(y_1), \ldots,f(y_n))=R(f(x_1), \ldots, f(x_n))$.

Supponiamo ora che la rete R non sia corretta, cioè che esista un vettore di interi $(x_1, \ldots,x_n)$ che tale che, dando in ingresso tale vettore alla rete R, R dia in uscita un vettore $(y_1,\ldots, y_n)$ non ordinato; si trovano allora due indici s,k per cui $y_s > y_k$ pur essendo $s<k$.

Consideriamo ora la funzione $g:N\rightarrow\{0,1\}$, definita da:

$$g(x) = \text{se } x \geq y_s \text{ allora 1, altrimenti 0}$$

Si osservi che $g(y_s)=1$ mentre $g(y_k)=0$.

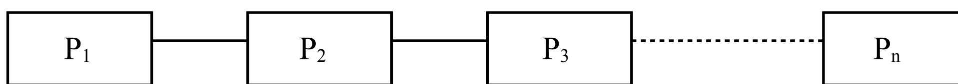
Poiché g è monotona, risulta $(g(y_1),\ldots,g(y_n))=R(g(x_1), \ldots, g(x_n))$: dando in ingresso a R il vettore $(g(x_1), \ldots, g(x_n))$ a componenti 0,1 si ottiene il vettore $(g(y_1),\ldots,g(y_n))$ che non è ordinato, poiché $g(y_s)=1$ mentre $g(y_k)=0$, con $s<k$.

Sulla base dei parametri che abbiamo introdotto, analizziamo e confrontiamo di seguito alcune topologie di reti di interconnessione:

- *Array lineare*
- *Mesh quadrata (o Array bi-dimensionale)*
- *Albero completo*
- *Ipercubo*

Array lineare

Un array lineare di dimensione n consiste di n processori connessi in modo sequenziale, cioe` il processore $P_i$ e` collegato a P_{i-1} e P_{i+1}, tranne i processori P_1 e P_N che sono, rispettivamente, collegati soltanto a P_2 e P_{N-1}.



Array Lineare di dimensione n.

Il grado di un array lineare e` 2, poiché ogni processore è collegato al più ad altri due.

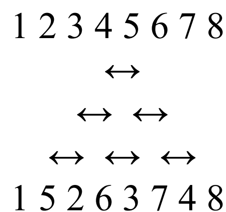
Il diametro di un array lineare è n-1, che rappresenta la distanza tra i processori più "lontani" P_1 ed P_n . Eseguire un algoritmo che richiede di processare informazioni inizialmente memorizzate in una qualsiasi coppia di processori richiede allora tempo $\Omega(n)$: in un array lineare risulta allora $\theta(n)$ il tempo necessario per determinare il massimo degli elementi di un vettore (A[1],…,A[n]), quando l'elemento A[i] è originariamente memorizzato nel processore P_i.

Se si elimina da un array lineare il lato che collega il processore $P_{n/2}$ al processore $P_{1+n/2}$, si ottengono due grafi disconnessi con lo stesso numero di vertici: l'ampiezza di bisezione di un array lineare è dunque 1: in un array lineare non è in generale possibile, in modo veloce, smistare agli altri processori informazioni originariamente memorizzate nei primi n/2 processori.

Prendiamo ora in considerazione due algoritmi paralleli realizzati su array lineare. Il primo, avendo in ingresso un vettore, ne calcola lo SHUFFLE, il secondo ordina il vettore stesso.

Esempio 5.1: algoritmo per il calcolo dello SHUFFLE

L'operazione SHUFFLE associa al vettore *(a[1], a[2], a[3], ..., a[2s]),* il vettore *(a[1], a[s+1], a[2], a[s+2], ...,a[s], a[2s]),* ottenuto alternando le componenti dei vettori *(a[1], a[2], ..., a[s])* e *(a[s+1], a[s+2], ..., a[2s]).*
Su un array lineare di n elementi, lo SHUFFLE di *(a[1], a[2], a[3], ..., a[2s])* può essere ottenuto in tempo O(s) da un procedura *SHUFFLE* con scambi a "triangolo" illustrata dalla seguente figura:



Esempio 5.2: algoritmo *P-D-ORD* su array lineare

Proponiamo ora un semplice algoritmo di ordinamento per confronti, che permette di ordinare con un array lineare un vettore di n elementi in tempo O(n): poiché per le considerazioni fatte sopra ogni algoritmo di ordinamento richiede tempo $\Omega(n)$, tale algoritmo è asintoticamente ottimo. Osserviamo che questo algoritmo è più veloce, sia pur di poco, di qualsiasi algoritmo per macchine a 1 processore, visto il tempo di un qualsiasi algoritmo sequenziale di ordinamento è $\Omega(n.\log n)$.

Consideriamo il problema di ordinare n elementi *a[1],..., a[n]* utilizzando n processori P_1,..,P_n con rete di interconnessione ad array lineare. Per ogni i, il processore P_i ha una variabile locale *A[i]* che prima dell'esecuzione dell'algoritmo contiene l'elemento *a[i]*; al termine dell'esecuzione le variabili *A[i]* , .., *A[n]* dovranno contenere gli elementi in ordine crescente.

Presentiamo ora l'algoritmo *P-D-ORD* . L'idea su cui si basa l'algoritmo è quella di dividere i passi di calcolo in pari e dispari; in un passo pari, gli elementi memorizzati nei processori di posto pari sono confrontati con quelli nel loro vicino sinistro ed eventualmente scambiati, e similmente nei passi dispari.

Una semplice sottoprocedura che utilizzeremo è *SCAMBIO(i,i+1)* , il cui risultato è lo scambio dei dati presenti in *A[i]* e *A[i+1]* tra i processori P_i e P_{i+1} .
Lo schema ad alto livello dell'algoritmo è il seguente:

ALGORITMO P-D ORD *(a[k] memorizzato in A[k] (1≤k≤n))*

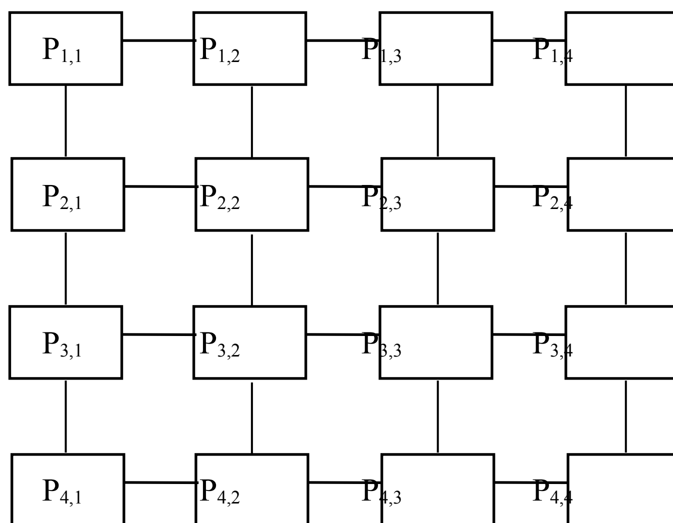
```
For 1≤t≤ n  do
    If  t    pari      then for 1≤k≤n/2 do in parallel
                              If A[2k-1]<A[2k] then SCAMBIA(2k-1,2k)
    If t dispari      then for 1≤k<n/2 do in parallel
                              If A[2k]<A[2k+1] then SCAMBIA(2k,2k+1)
```

L'idea della procedura è di fare in modo che, nei passi pari, i processori P_1 e P_2, P_3 e P_4, e così via, si scambino i dati presenti in *A[2i-1]* e *A[2i]* solo se *A[2i-1] <A[2i]*; nei passi dispari i processori P_2 e P_3, P_4 e P_5, e così via, fanno la stessa cosa. L'algoritmo è facilmente implementabile, usando un'altra variabile locale in ogni processore, perché i test sono effettuati su variabili di processori adiacenti.

Si osservi che il dato minore, ad ogni passo, si "muove" da un processore al precedente finchè non arriva alla posizione 1; il dato penultimo in ordine fa la stessa cosa, salvo in un eventuale passo in cui incontra il dato minore, e così via. Questo significa che, dopo al più n passi, tutti i dati si troveranno nella corretta posizione: il k° dato dal basso si troverà nel processore P_k .

Mesh quadrata

E' la versione bidimensionale dell'array lineare. Consiste in $n = m^2$ processori inseriti in un griglia $m\times m$, tale che il processore $P_{i,j}$ e` connesso ai processori $P_{i\pm 1,j}$ e $P_{i,j\pm 1}$ (salvo ai confini).



mesh 4x4

Il grado della mesh quadrata è dunque 4, che è il grado dei vertici interni.

I processori più "lontani" sono P_{11} e P_{mm}:essi sono collegati da un cammino di 2(m-1) passi, quindi il diametro di una mesh è $2\cdot (\sqrt{n} - 1)$.

Ogni algoritmo che richiede di processare informazioni originariamente memorizzate in una qualsiasi coppia di processori richiede allora un tempo di esecuzione $\Omega(\sqrt{n})$: si osservi che questo limite è inferiore a quello ottenuto per l'array lineare ($\Omega(n)$), ma decisamente superiore a prestazioni ottenute per vari problemi su PRAM ($O(\log n)$).

Per dividere una mesh quadrata in due rettangoli disconnessi di ugual numero di vertici basta "tagliare" le connessioni tra i processori $P_{m/2\ k-1}$ e $P_{m/2\ k}$ (k=1,m/2): l'ampiezza di bisezione di una mesh mxm è allora $\sqrt{n}$, e il tempo di trasferimento di dati memorizzati in un rettangolo nell'altro non potrà essere inferiore a $\sqrt{n}/2$. Questo implica, in particolare, che ogni algoritmo di ordinamento su mesh richiede almeno $2\sqrt{n}$ passi di calcolo.

Esempio 5.3: Determinazione del MASSIMO su mesh

Consideriamo il problema MASSIMO su una mesh mxm. Supponiamo che ogni processore P_{ij} abbia una variabile locale *A[i,j]* che inizialmente contiene la componente *a[i,j]* della matrice a= (*a[i,j]*) di cui vogliamo calcolare il massimo. Al termine dell'esecuzione della procedura, si vuole che $M=\max_{ij} a[i,j]$ sia memorizzato nel processore P_{11}.

Per disegnare la procedura basta osservare che una mesh quadrata mxm può essere vista, dimenticando le connessioni verticali, come una collezione di m array lineari di m processori (righe). Analogamente, dimenticando le connessioni orizzontali, può essere vista come una collezione di m array lineari di m processori (colonne).

La procedura, ad alto livello, è la seguente:

ALGORITMO "MAX Mesh"

- *Trova il massimo di ogni riga i, mettendo il risultato in A[i1].*
- *Trova il massimo sulla prima colonna, mettendo il risultato in A[11].*

Entrambe i passi, agendo in parallelo sulle righe (colonne) viste come array lineari, richiedono tempo $O(\sqrt{n})$, che risulta asintoticamente ottimo.

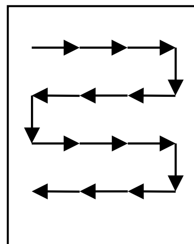
Esempio 5.4: algoritmo *SortMergeLS3* su mesh

Presentiamo qui un algoritmo che risolve su mesh il problema ORDINAMENTO in tempo ottimo $O(\sqrt{n})$; per semplicità, prenderemo in considerazione mesh mxm dove m è potenza di 2.

Supponiamo per prima cosa che il vettore da ordinare *(a[1],.., a[n]),* con $n=m^2$, si trovi memorizzato “a serpente” nella mesh mxm. Questo significa:

- se i è pari, nella variabile *A[ik],* locale al processore P_{ik} , si trova memorizzato il numero *a[(i-1).m+k)].*
- se i è dispari, nella variabile *A[ik],* locale al processore P_{ik} , si trova memorizzato il numero *a[(i-1).m+m-k+1]*.

Qui sotto viene mostrata la memorizzazione a “serpente” di un vettore di 16 componenti in una mesh

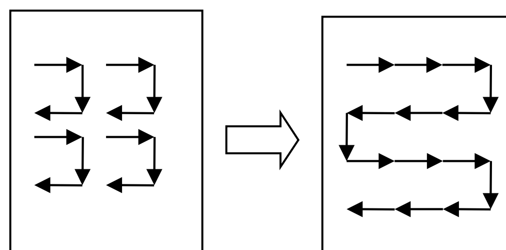


Vettore “ a serpente” in mesh

Obbiettivo di un algoritmo di ordinamento su mesh è di fare in modo che, dopo l’esecuzione della procedura, il vettore si trovi memorizzato a “serpente” ordinato in modo crescente. Si osservi che i processori ordinati “a serpente” formano un array lineare, è quindi possibile applicare l’algoritmo *P-D-ORD* precedentemente visto; tale algoritmo tuttavia ha un tempo di calcolo O(n), non sfruttando altre potenziali connessioni tra i processori.

Discutiamo ora l’algoritmo LS3, dovuto a Lang, Schimmler, Schmeck e Schroeder (1985), che si basa sulla tecnica “divide et impera” e risolve (come nel caso del Bitonic Sort su P-RAM) il problema richiamando una procedura di Merge.

Ogni mesh mxm può essere suddivisa in 4 mesh m/2 x m/2 . La procedura di *MergeLS3* ordina a serpente, sulla mesh mxm, un vettore che in partenza è ordinato, a serpente, in ognuna delle 4 sottomesh, come rappresentato nella figura seguente (relativa al caso m=4):

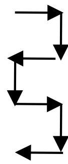


Effetto della procedura di Merge

Questo algoritmo richiama le procedure di *SHUFFLE* e *P-D-ORD,* descritte nella sezione dedicata all'array lineare, applicate ad array lineari che sono opportuni sotto-array della mesh.

Per prima cosa, ricordiamo che ogni riga della mesh mxm può essere vista come un array lineare di m processori. Chiameremo *SHUFFLE (i)* l'applicazione della procedura *SHUFFLE* alla riga i della mesh.

Osserviamo poi che due colonne consecutive di processori di una mesh possono essere visti come un array lineare se si ordinano i processori "a serpente", come illustrato della figura seguente



Chiameremo *DOPPIA-COLONNA(i,i+1)* la procedura che si ottiene applicando l'algoritmo *P-D-ORD* ai processori delle colonne i e i+1 della mesh, che formano un array lineare nell'ordinamento "a serpente".

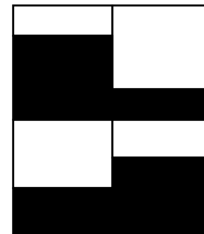
La procedura *MergeLS3* può essere così descritta:

Procedura MergeLS3 (m x m mesh in cui le 4 m/2 x m/2 sottomesh sono già ordinate a serpente)

1. *Applica SHUFFLE(i) in parallelo per 1≤i≤m (sulle righe)*
2. *Applica DOPPIA-COLONNA(2i-1,2i) in parallelo per 1≤i≤m/2*
3. *Applica i primi 2m passi di P-D-ORD all'array lineare formato dal serpente della mesh.*

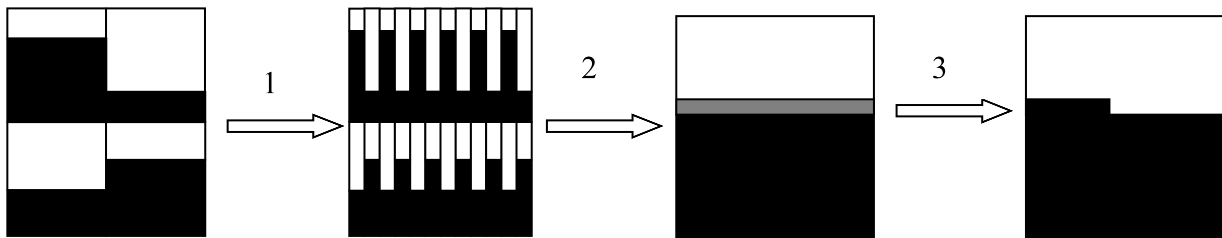
Per provare, sia pur a grandi linee, la correttezza della procedura *MergeLS3*, ricorriamo al "*principio 0,1*" . Prendiamo in considerazione quindi vettori memorizzati in mesh a componenti 0 o 1; un vettore ordinato "a serpente" darà luogo ad una mesh con righe superiori a componenti 0, righe inferiori a componenti 1, salvo eventualmente una riga di confine che può contenere sia 0 che 1. La seguente figura rappresenta a sinistra un vettore ordinato in mesh mxm, a destra un vettore che risulta ordinato in ognuna delle quattro sottomesh m/2 x m/2 (le componenti 0 sono rappresentate in bianco, le componenti 1 in nero, per semplicità non è rappresentata la riga di confine)

Vettore ordinato



Vettore ordinato nelle sottomesh

La procedura *MergeLS3* lavora nei tre passi come segue:



Il primo passo (*SHUFFLE* per riga) fa in modo che le coppie di colonne adiacenti siano uguali (a parte le righe di confine). Il secondo passo (*DOPPIA COLONNA*) ricostruisce il vettore ordinato, a parte la riga di confine. Bastano infine 2m passi per ordinare la riga di confine.

Ognuno dei tre passi della procedura *MergeLS3* lavora in tempo O(m), quindi l'intera procedura ha un tempo di calcolo al più c $\sqrt{n}$, per una opportuna costante c.

Possiamo ora descrivere la procedura *MergeSortLS3* che, nei suoi passi essenziali, risulta essere:

Procedura MergSortLS3 (m; vettore (a[1],..,a[n]) memorizzato a serpente)
Se m=1 allora return a
altrimenti

- *dividi la mesh mxm in 4 sottomesh m/2 x m/2*
- *applica ricorsivamente in parallelo la procedura di MergeSortLS3 a ognuna delle sottomesh m/2 x m/2.*
- *applica la procedura di MergeLS3 alla mesh mxm con la configurazione ottenuta.*

La correttezza è immediata poiché in ingresso alla procedura *MergeLS3* risulta un vettore ordinato nelle quattro sottomesh m/2 x m/2.

Analizziamo ora il tempo di calcolo. Detto $T_{MS}(n)$ il tempo per ordinare un vettore di n elementi con la procedura *MergeSortLS3* e $T_M(n)= c\sqrt{n}$ il tempo necessario a ordinare con la procedura *MergeLS3* un vettore parzialmente ordinato di n elementi, vale:

$$T_{MS}(n) \leq T_{MS}(n/4) + T_M(n) = c\sqrt{n} + T_{MS}(n/4)$$

Pertanto:

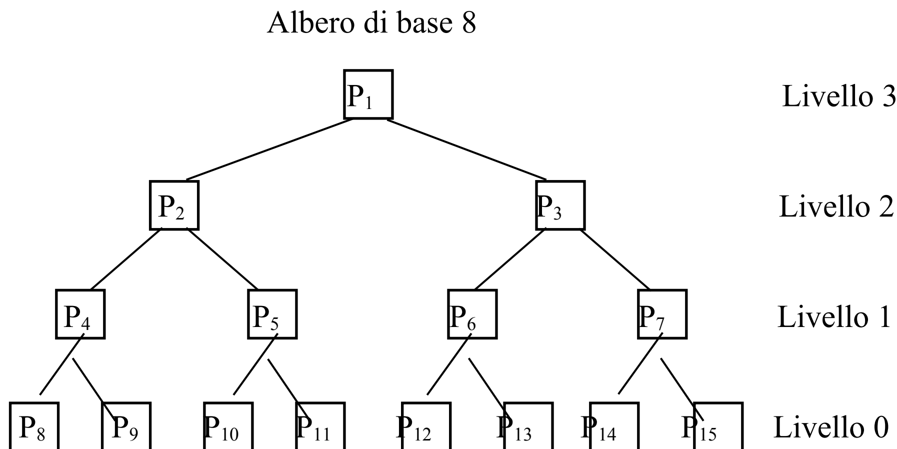
$$T_{MS}(n) \leq c\sqrt{n} + c\sqrt{n/4} + c\sqrt{n/16} + \ldots \leq 2c\sqrt{n}$$

Concludiamo che la procedura *MergSortLS3* ordina un vettore di n elementi in tempo asintoticamente ottimale $O(\sqrt{n})$.

Albero di base n

Un albero di base n è un albero binario completo con n foglie; i processori sono posti ai nodi dell'albero. L'albero e` composto da *d* livelli, numerati da 0 a d-1, quindi $n=2^{d-1}$ e il numero di processori è 2n-1.

Albero di base 8



Ogni processore P_k , che non sia radice o foglia, e` connesso sia al processore padre (al livello d+1) che ai due figli (al livello d-1), mentre la radice non ha padre e le foglie non hanno figli: l'albero di base n ha quindi grado 3.

Osserviamo ora che il cammino da una foglia alla radice richiede d-1 passi: il diametro e` allora pari a $2(d-1) = \theta(\log n)$. Questo basso diametro permette di risolvere in maniera efficiente il problema Massimo

Esempio 5.5: determinazione del MASSIMO

Il seguente algoritmo risolve in tempo asintoticamente ottimo $O(\log n)$ il problema di determinare il massimo elemento in un vettore $(A_1,\ldots, A_n)$, le cui componenti sono inizialmente memorizzate nelle foglie dell'albero di base n.

For all k (k=1,n) do in parallel
    "il processore P_k posto alla foglia k invia A_k al proprio padre"
For t=2,log n –1 do
        For all "nodo interno v" do in parallel
            If " P_v posto al nodo v riceve a dal figlio sinistro e b dal figlio destro" then
                    $c_v = Max\{a,b\}$
                    P_v invia c_v al proprio padre
            If " P_r posto alla radice r riceve a dal figlio sinistro e b dal figlio destro" then
                    $c_r = Max\{a,b\}$

L’esecuzione termina dopo O(log n) passi di calcolo e al termine dell’esecuzione il valore del massimo è memorizzato nella radice.

Analizziamo ora l’ampiezza di bisezione dell’albero di base n. Tagliando le due connessioni tra la radice e i suoi figli, si ottengono due alberi disconnessi di base n/2: l’ampiezza di bisezione dell’albero è dunque 2.

Questa bassa ampiezza di bisezione è causa di colli di bottiglia nell’esecuzione di algoritmi che richiedono di trasferire grandi quantità di dati. Per esempio, in un qualsiasi algoritmo di ordinamento si devono poter trasferire gli n/2 dati memorizzati nel sottoalbero di sinistra nel sottoalbero di destra: questi dati devono passare per la radice e il processore P_r può trasferire al più un dato alla volta. Concludiamo col seguente risultato negativo:

Fatto 5.4: ogni algoritmo di ordinamento su un albero di base n richiede tempo $\Omega(n)$.

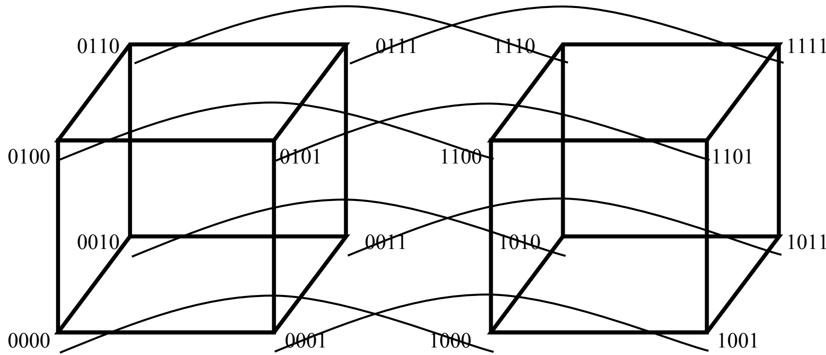
In conclusione, il basso diametro dell’albero permette la soluzione efficiente di alcuni problemi che richiedono di combinare informazioni distribuite nell’albero, a patto che non vengano smistate da una parte all’altra grandi quantità di dati: in tal caso la bassa ampiezza di bisezione è responsabile di prestazioni povere.

Ipercubo

Per $n = 2^d$, un ipercubo (o d-cubo) è un grafo i cui vertici sono elementi $c_1 \dots c_d \in \{0,1\}^d$ e due vertici x e y sono estremi di un lato se le parole x e y differiscono in una sola posizione: in una architettura a ipercubo i processori P_x e P_y sono quindi collegati se x e y differiscono in una sola posizione. Per esempio, il processore P_{1001} e\` collegato a P_{1000} ma non a P_{1111}

L’ipercubo e\` una struttura ricorsiva: un d+1-cubo può essere ottenuto connettendo due d-cubi. Uno dei due cubi possiede i processori i cui indici hanno il bit piu\` significativo pari a 0, mentre l’altro cubo quelli i cui indici hanno il bit piu\` significativo uguale ad a 1.

Un 4-cubo



Ogni vertice $c = c_1 \dots c_d$ è collegato direttamente ai d vertici ottenuti modificando un bit di c: il grado di un d-cubo è dunque d = log n.

I vettori più distanti di un cubo sono 00..0 e 11..1: essi sono collegati da un cammino di lunghezza d, quindi il diametro di un d-cubo è d = log n. Il basso diametro di un d-cubo permette di risolvere in modo efficiente problemi che richiedono di combinare informazioni distribuite. In particolare, non è difficile progettare algoritmi in tempo O(log n) per la soluzione del problema MASSIMO. Presentiamo qui un semplice algoritmo per tale compito.

Esempio 5.6: Determinazione del MASSIMO.

Dato il vettore $(A_1,\dots, A_n)$, supporremo che la componente A_k sia memorizzata, inizialmente, nella memoria locale A[k] del processore P_k (k=1,n) (qui con abuso di notazione identifichiamo l'intero k con la sua rappresentazione binaria); il risultato della computazione sarà il massimo $Max_k\ A_k$ memorizzato in A[0] nel processore P_0 (si noti che gli indici dell'array iniziano da 0). Nell'algoritmo che segue, $i^{(b)}$ è ottenuto complementando il *b-esimo* bit dell'indice *i* in notazione binaria; ad esempio, $1000110^{(3)}$ è 1010110. Si osservi in particolare che il processore P_i e $P_i^{(b)}$ sono collegati.

ALGORITMO (Massimo su un ipercubo)
Input: Un array $(A_1,\dots, A_n)$ dove A(i) e` memorizzato nella memoria locale A[i] del processore P_i (i=1,n).

For b= d-1, 0 do in parallel
if $(0 \le i \le 2^b-1)$ $\begin{cases} P_i^{(b)} \text{ invia a } P_i \text{ il contenuto di } A[i^{(b)}] \\ A[i] = Max\ \{A[i], A[i^{(b)}]\} \end{cases}$

La correttezza si prova osservando che, posto $M= Max_k\ A_k$, quando b assume valore s, con $0\le s<d$, il massimo dei valori contenuti in A[i] con $0 \le i \le 2^b-1$ è M. In particolare, all'ultimo passo vale b=0, ne segue che $0 \le i \le 2^b-1$ implica i=0 e quindi il processore P_0 contiene M in A[0].

Per disconnettere i due d-1-cubi con vertici 0x e 1y è necessario tagliare n/2 lati di un d-cubo. Più in generale, per disconnettere due sottonisiemi disgiunti A e B del d-cubo, ognuno di cardinalità n/2, è necessario in generale "tagliare" almeno n/2 lati: l'ampiezza di bisezione di un d-cubo risulta allora n/2. Se l'alta ampiezza di bisezione rende difficile la realizzazione h/w di una rete a ipercubo, la stessa permette di smistare in parallelo grandi quantità di dati. Ad esempio, contrariamente a quanto visto per l'albero di base n, sull'ipercubo il problema dell'ordinamento può essere risolto efficientemente in tempo $O(\log^2 n)$ implementando opportunamente l'algoritmo bitonico, presentato per le P-RAM.

ALCUNI RIFERIMENTI BIBLIOGRAFICI

Un libro dedicato allo studio del disegno e analisi degli algoritmi probabilistici è

- R. Motwani and P. Raghavan. Randomized Algorithms. Cambridge University Press, New York (NY), 1995.

Tale libro è interessante, ma di non facile lettura. Una più semplice introduzione agli algoritmi probabilistici sono il Cap. 11 del classico libro di complessità computazionale

- Christos Papadimitriou , *Computational Complexity*, 1st edition, Addison Wesley, 1993.

e, con riferimento particolare ai protocolli di comunicazione

- J. Hromkovic, Communication protocols: an exemplary study of the power of randomness. Handbook of Randomized Computing, Vol.2, pp.533-596, Kluwer Academic Publishers, 2001.

L'algoritmo di Rabin è stato pubblicato su

- M. O. Rabin. (1980), "Probabilistic Algorithm for Testing Primality." *Journal of Number Theory* 12:128-38.

Può essere interessante ricordare che già nel 1874 un libro dell'economista e filosofo inglese William Stanley Jevons descrive la relazione tra funzioni one-way e la crittografia.

L'articolo originale sul crittosistema RSA è

- Rivest, R.; A. Shamir; L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM* **21** (2): pp.120-126, 1978.

Può essere interessante osservare che i primi esperimenti di simulazione al computer di "vita artificiale" sono dovuti al matematico italo-norvegese Nils Aall Barricelli (1912-1993), a Princeton, New Jersey nel 1953.

Una introduzione ai modelli evolutivi, con presentazione di tecniche di analisi elementari, si trova in

- Holland, Adaptation in Natural and Artificial Systems, The MIT Press; Reprint edition 1992 (originally published in 1975).

Il dilemma del prigioniero è stato introdotto come esempio di giochi economici negli anni '50. E' stato ripreso da Robert Axelrod nel suo libro "The Evolution of Cooperation " (1984), in cui si evidenzia l'importanza di tecniche evolutive per l'individuazione di buone strategie.

La ricerca locale è una delle tecniche più diffuse e note. Un volume sull'argomento è, ad esempio

- E. Aarts, J. K. Lenstra, Local Search in Combinatorial Optimization, Princeton University Press, 2003

L'idea dell'algoritmo di Metropolis si trova esposta per la prima volta in

- N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. "Equations of State Calculations by Fast Computing Machines". Journal of Chemical Physics, 21(6):1087-1092, 1953.

L'estensione all'Annealing simulato è presentata in

- S. Kirkpatrick and C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, Science, Vol 220, Number 4598, pages 671-680, 1983

Un testo introduttivo con applicazioni a vari problemi di ottimizzazione combinatoria è per esempio

- E. Aarst. J . Korst: "Simulated Annealing and Boltzmann Machines". Chichester, Wiley, 1989

Le idee di base sui principi per l'ordinamento parallelo (tra cui l'algoritmo P-D-Ord e il teorema **"*principio 0,1*")** si possono trovare in:

- D.E. Knuth, The Art of Computer Programming, Vol. 3 - Sorting and Searching. Addison-Wesley (1973)

L'algoritmo "Bitonic sort" è pubblicato in

- K.E. Batcher, Sorting Networks and their Applications. Proc. AFIPS Spring Joint Comput. Conf., Vol. 32, 307-314 (1968)

L'algoritmo di ordinamento in mesh SortLS3 è pubblicato in

- H.W. Lang, M.Schimmler, H. Schmeck, H. Schroeder, Systolic Sorting on a Mesh-Connected Network. IEEE Transactions on Computers C-34, 7, 652-658 (1985)

Un testo recente stimolante su nuovi aspetti nell'area degli algoritmi è:

- J. Hromkovic, Algorithmic Adventures: from knowledge to Magic, Springer, 2009.