Algoritmi Randomizzati o Probabilistici

A.A. 2017-2018

Algoritmi randomizzati

Borel: ***"un evento che ha probabilità minore di 10^{-50} non accadrà mai e, se dovesse accadere, non potrebbe mai essere osservato"***

Obiettivo: Accelerare la rapidità di una risposta sacrificandone la precisione.
Se l'errore non è troppo probabile, o comunque tollerabile in ragione del risparmio di tempo, la scelta può rivelarsi buona.

La nozione di algoritmo probabilistico può essere formalizzata in termini della nozione di algoritmo non deterministico, interpretato in modo diverso.

La computazione di un algoritmo non deterministico è caratterizzata da un albero delle scelte dove ogni percorso radice-foglia corrisponde a una sequenza di scelte, cioè a una computazione di una copia dell'algoritmo.
Una scelta viene effettuata tramite la ***choice*** {I} che sceglie arbitrariamente un elemento dell'insieme I.

Algoritmi randomizzati

La computazione restituisce SI solo se esiste una sequenza di scelte che porta al successo cioè se almeno una foglia dell'albero delle scelte esegue una ***success*** mentre dà risposta NO se tutte le scelte portano al fallimento, cioè se tutte le foglie dell'albero eseguono ***failure***.

Per passare al probabilismo bisogna reinterpretare la modalità di ogni scelta e ridefinire il risultato di una computazione nell'albero delle scelte.

Una scelta non è più effettuata **non deterministicamente**, ma **probabilisticamente** tramite la funzione ***RANDOM*** (A) che restituisce con la stessa probabilità un qualsiasi elemento scelto casualmente nell'insieme A. I figli di un nodo nell'albero delle scelte sono visti come possibili alternative equiprobabili anziché come scelte non deterministiche.

Algoritmi randomizzati

Otteniamo quindi una simulazione probabilistica dell'algoritmo non deterministico, che equivale a scegliere a caso nell'albero delle scelte, producendo in uscita il valore ottenuto dall'esecuzione.

Questi algoritmi vengono chiamati ***randomizzati*** o ***probabilistici***, (o ***stocastici***) e li descriveremo nel linguaggio di disegno usato finora, con l'aggiunta dell'istruzione random(A), con la semantica prima descritta.

Un algoritmo deterministico calcola una funzione f(x) se associa al valore di ingresso x il valore di uscita y = f(x).
È possibile estendere tale nozione al caso probabilistico?

Per definizione gli algoritmi probabilistici lasciano alcune decisioni al caso. La loro caratteristica fondamentale è che possono reagire in modo diverso quando vengono applicati due volte alla stessa istanza. Il tempo d'esecuzione e anche il risultato ottenuto possono considerevolmente variare da un'esecuzione all'altra.

Esempio: Protocollo di comunicazione randomizzato per l'uguaglianza

Due ricercatori R_1 ed R_2 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 R_1 conterrà la parola binaria $a = a_1,.., a_n$, quello di R_2 la parola binaria $b = b_1,.., b_n$.
Naturalmente 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:

R_1 invia $a = a_1,.., a_n$ a R_2
R_2 riceve a da R_1 e lo confronta con b; se $a = b$ risponde successo, altrimenti risponde fallimento .

Osserviamo che il protocollo in questione è deterministico, e prevede lo scambio di n 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:

- R_1 calcola s = random ($\{p \mid p$ primo, $2 \le p \le n^2\}$ e $z = a \bmod s$
- R_1 invia a R_2 la coppia $<s,z>$
- R_2 riceve da R_1 la coppia $<s,z>$ e calcola $b \bmod s$; se $b \bmod s = z$ allora risponde successo, altrimenti risponde fallimento.

Quanti bit R_1 invia a R_2?

Poiché $z < s \le n^2$, sia s che z sono esprimibili con al più

$$\log n^2 = 2 \log n \text{ bit.}$$

R_1 invia dunque $4 \log n$ bit a R_2.

Il protocollo è probabilistico, quindi deve essere prevista la possibilità di errore.
Qual è la probabilità di errore?

E' facile convincersi che l'unico caso di errore si ha quando
$a \neq b$ ma $a \bmod s = b \bmod s$.

Senza perdita di generalità, supponiamo $a > b$;
posto $a - b = v$, sarà $v < 2^n$, $v \neq 0$, ma $v \bmod s = 0$.

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/ln N, si ha:

$$\#\text{Casi possibili} = n^2/\ln n^2.$$

Il numero di casi favorevoli è il numero di divisori primi di *v*;
decomponendo v in fattori primi si ha:

$$2^n > v = p_1^{J1} \ldots p_t^{J\,t} > 2^t$$

e quindi, essendo il numero t di fattori primi di *v* al più n, otteniamo:

$$\#\text{Casi ''favorevoli''} < n$$

Possiamo concludere che la probabilità di errore è al più $2 \cdot \ln n/n$.

Esempio: Numeri primi

Il problema puo` essere risolto con il seguente algoritmo deterministico

Primo (n)
    while ($\lfloor j <= \sqrt{n} \rfloor$)
        if (n mod j = 0) return *false*
        j ← j+1
    return *true*

La complessita` e` $\mathbf{O}(n^{1/2})$, che non e` polinomiale nella dimensione del problema, che e` d = log n + 1.
La complessita` di "*Primo*" e` pertanto esponenziale nella dimensione del problema.

Esempio: Numeri primi

Primalità $\in$ **NP**
Non-primalità $\in$ **NP**

Nel 2002 è stato dimostrato che il problema Primalità è in **P**.
AKS (Agrawal, Kayal, Saxena): algoritmo deterministico polinomiale ($\mathbf{O}((\log_2 n)^{12+\varepsilon})$; una variante proposta nel 2005 da Pomerance e Lenstra $\mathbf{O}((\log_2 n)^{6+\varepsilon})$).

Nelle situazioni pratiche si preferiscono algoritmi non precisi, che a volte falliscono, ma più rapidi.

I rischi devono essere bilanciati da accelerazione sui tempi rispetto agli algoritmi deterministici “sicuri”.

N.B. Si suppone $A \in$ **NP** and $A^C \in$ co-**NP** $\cap$ **NP** sia indicazione del fatto che $A \in$ **P**

Proprietà dei numeri primi

Per i numeri primi n > 2 valgono le proprietà interessanti. La più famosa è forse questa:

"**Piccolo" teorema di Fermat**: se n è primo per ogni $2 \leq r < n$

$$r^{n-1} \textit{mod}\ n = 1. \qquad (1)$$

La L'equazione (1), non vale per quali tutti i numeri composti, ma non per tutti. Per esempio 4^{14} *mod* 15 = 1, ma 15 non è primo.

In particolare vi sono dei numeri composti (numeri di *Carmichael*) che soddisfano l'equazione (1) per ogni r, ma sono pochissimi (solo 225 neo primi 10^8 numeri). I primi sono 561, 1105, 1729)

Diciamo che n è uno *pseudoprimo* (di Fermat) rispetto ad r se $r^{n-1} \textit{mod}\ n = 1$.

Altre Proprietà dei numeri primi

Per i numeri primi n > 2 vale anche la **proprietà** seguente:

Sia n un numero primo e r un intero compreso tra 1 e n -1.
Le uniche soluzioni di $r^2 mod\ n = 1$ sono $r\ mod\ n = 1$ e $r\ mod\ n = -1$.

Tale proprietà non è sempre vera per i numeri composti.

Definizione. Sia ora $n-1 = 2^s \cdot d$, con d dispari e n pseudoprimo per la base r.
Allora n è uno *pseudoprimo forte* per la base r se nella sequenza di valore
$r^d \bmod n, r^{2.d} \bmod n, .. r^{2^s.d} \bmod n$
l'ultimo valore è 1 e l'ultimo valore diverso da 1 può essere solo -1

Teorema (Miller). Se è valida l'ipotesi di Riemann estesa **n** è primo se e solo e è pseudoprimo forte per ogni base $< 2 \cdot (\log n)^2$.
In tal caso *primalità* sarebbe dimostrata in **P.**

(1) $p^a \bmod n = 1 \Rightarrow p^{a.2} \bmod n = 1$. Infatti $p^{a.2} \bmod n = (p^a \bmod n)^2 \bmod n = 1 \bmod n$

Algoritmo di Rabin-Miller

Posto $n-1 = 2^s \cdot d$: si calcolano i quadrati, r^{2d}, r^{4d}, ….$r^{2^{s-1}.d}$ di cui r^d, r^{2d}, ….$r^{2^{s-2}.d}$ sono le radici quadrate. Se il primo di questi numeri che vale 1 non è preceduto da un -1 (o se n non soddisfa il test di Fermat), r è un testimone che n non è primo.

La procedura WITNESS cerca un testimone che n *non* è primo

```
WITNESS (r, n)   \\ r è un testimone che n è composto
    siano  s e t tali che d è dispari e n-1 = 2^s · d
    x_0 = r^d mod n
    for i =1 to t
          x_i = x^2_{i-1} mod n
          if x_i = 1 and x_{i-1} ≠ 1 and x_{i-1} ≠ -1   \\ -1 mod n = n-1 mod n
                return TRUE
     if x_t ≠ 1 return TRUE      \\ x_t è r^{n-1} mod n
     return FALSE
```

Algoritmo di Rabin-Miller

s e d si possono calcolare dimezzando n-1 per $\mathbf{O}(\log n)$ volte e il calcolo di r^d *mod* n si puo` fare in tempo $\mathbf{O}(\log n)$ usando algoritmi di esponenziazione modulare veloce, analogamente le potenze $r^{2^i \cdot d}$ *mod* n si calcolano per quadrature successive di r^d*mod* n.

```
modpot (a,b,c)          \\calcola a^b mod c
    if b = 1 return (a mod c)
    else   if (b pari) z ← modpot (a, b/2, c)
                       return (z^2 mod c)
           else z ← modpot (a, b -1, c)
                  return ((a·z) mod c)
```

Se si trova un valore diverso da 1 e il primo di questi è diverso da -1, si può dedurre che n è composto, altrimenti n è **probabilmente primo**

Prendiamo r=2

N = 13
N-1 = $2^2 \cdot 3$
$2^{2^2 \cdot 3} = 2^{12}$
2^{12} mod 13 = 1
$2^{2.3} = 2^6 = 64 = 65-1$
65-1 *mod* 13 = -1

13 è "probabilmente primo"

N = 7
N-1 = $2 \cdot 3$
$2^{2 \cdot 3} = 2^6$
2^6 *mod* 7 = 1
2^3 mod 7 = 1

7 risulta "probabilmente primo"

Vediamo un caso più critico:
N = 561 = 3 · 11 · 17 (numero di Carmichael) 560 = $2^4 \cdot 35$
Per qualunque b primo con 561 si ha $b^{560} mod\ 561 = 1$
poiché 561 è pseudoprimo per b.
Prendiamo $r = b^{2^4} = b^{16}$:

$r^{2^4.35} = r^{560} = (b^{560})^{2^4}$	$(b^{560})^{2^4} mod\ 561 = 1$
$r^{2^3.35} = r^{280} = (b^{560})^{2^3}$	$(b^{560})^{2^3} mod\ 561 = 1$
$r^{2^2.35} = r^{140} = (b^{560})^{2^2}$	$(b^{560})^{2^2} mod\ 561 = 1$
$r^{2.35} = r^{70} = (b^{560})^2$	$(b^{560})^2 mod\ 561 = 1$
$r^{35} = b^{560}$	$b^{560} mod\ 561 = 1$

561 è pseudiprimo forte rispetto a r, ma **non** rispetto ad altri valori di r

$r = b^{2^4}$ è testimone inattendibile della primalità di 561 per ogni b primo con N.

Prendendo invece r = 2 l'algoritmo risponde con certezza che 561 non è primo.

$$2^{560} \textit{mod}\ 561 = 1$$
$$2^{280} \textit{mod}\ 561 = 1$$
$$2^{140} \textit{mod}\ 561 \neq +/- 1$$

Si può dimostrare che per ogni N composto c'è almeno un testimone onesto r che lo dichiara tale, anzi, almeno ¾ degli interi r interpellati è attendibile e quindi la probabilità di errore per un singolo r è al massimo ¼.

Il ricorso a k testimoni scelti in modo casuale una loro concorde risposta abbassano la probabilità di errore che diventa <= $1/4^k$.

Se k = 6 $1/4^k = 0{,}0002441$

```
PRIMO (n)
    for j ← 1 to 50
        r ← random (2, n-1)
        if (r testimonia che n è composto)
                return false
    return true
```

Esempio: 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 $2 \leq p \leq x$. Una soluzione può essere:

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

```
for j ← 1 to n
        y_j ← random ({0, 1})
y ← y_1,…, y_n
while (x < y o y è composto)
        for j ← 1 to n
                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 possiamo usare 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 solo approssimativamente x/ln x numeri primi minori di 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 atteso di ln x iterazioni. Poiché $x < 2^n$, ln x = $\mathbf{O}(n)$, quindi l'algoritmo ha un tempo atteso polinomiale.

Algoritmi randomizzati

Nel 1977 Gill ha introdotto una sequenza di classi probabilistiche di complessità, dove ogni classe è ottenuta dalla precedente con l'imposizione di vincoli.

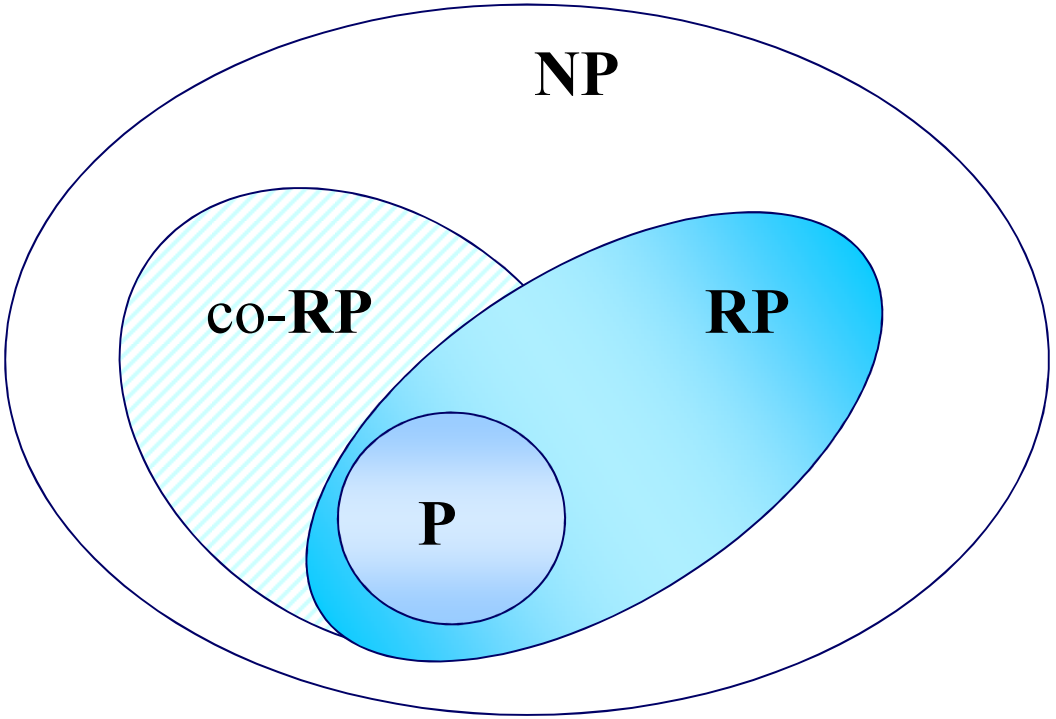
Classe **RP** (random polinomiale): l'albero delle scelte non deve avere alcuna foglia che ha *fallimento* se la risposta è SI.
Gli algoritmi per i problemi della classe **RP** danno risposta SI con probabilità di errore nulla.
L'errore può presentarsi per la risposta NO con probabilità $\leq 1/2$, ma la probabilità di errore può essere resa arbitrariamente piccola con un numero polinomiale di iterazioni.

- **RP** non è chiusa rispetto al complemento
- **RP** $\subseteq$ **NP**

- **P** $\subseteq$ **RP** $\cap$ co-**RP**
- Nessun problema **NP**-completo può essere in **RP**, a meno che **RP** = **NP**.

Non-Primalità $\in$ **RP**, non si sapeva se Primalità $\in$ **RP**

Algoritmi randomizzati



Se l'algoritmo risponde "successo", la risposta è corretta, se invece risponde "fallimento" allora la probabilità d'errore è al massimo 1/2.
Questa probabilità 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 un'esecuzione di A su x si ottiene il seguente algoritmo probabilistico:

```
for k ← 1 to t
      R[k] ← A_random (x)
      if R[k] = "successo"  return successo
return fallimento
```

La probabilità d'errore è limitata in questo caso da $(1/2)^t$. se δ è il livello di rischio ritenuto accettabile, basta scegliere t in modo che $1/2^t = \delta$, cioè $t = \lg(1/\delta)$.
La crescita di t è molto contenuta in termini di δ, basta ripetere 10 volte l'algoritmo per ridurre la probabilità d'errore a 10^{-3}.

Una metodologia molto usata per disegnare algoritmi probabilistici è il "**fingerprinting**". Supponiamo di voler decidere se due elementi x e y di grande dimensione sono diversi, avendo a disposizione una famiglia H di funzioni.
Si può usare il seguente algoritmo randomizzato:

h ← random (H)
if h(x) ≠ h(y) return successo
else return fallimento

Se la funzione h è calcolabile in modo efficiente 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 successo, e quindi h(x) ≠ h(y), risulterà x ≠ y. In caso contrario esiste la possibilità di errore e la classe H deve essere scelta attentamente in modo che la probabilità d'errore sia al massimo ½.

Esempio: Non Commutatività di matrici

Date due matrici di interi A e B: n x n, verificare se esse non commutano, cioè se $A \cdot B \neq B \cdot A$.

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

Un modo per comprimere una matrice A: n x n è quello di scegliere a caso un vettore X: 1 x n in {-1, 1} e calcolare $Z = X \cdot A$; in questo caso il calcolo avviene in $\mathbf{O}(n^2)$ passi e l'***impronta*** di A è il vettore Z: 1 x n.

Grazie alla proprietà associativa del prodotto di matrici rettangolari, vale:

$$X\cdot(A\cdot B) = (X\cdot A)\cdot B \quad , \quad X\cdot(B\cdot A) = (X\cdot B)\cdot A$$

Questo dà luogo al seguente algoritmo randomizzato per decidere se $A\cdot B \neq B\cdot A$:

Commuta$_{\text{RANDOM}}$(due matrici n×n A e B)
  X ← random ({-1,1})
  $V_1 \leftarrow X\cdot A$
  $V_2 \leftarrow V_1\cdot B$
  $Z_1 \leftarrow X\cdot B$
  $Z_2 \leftarrow Z_1\cdot A$;
  if ($V_2 \neq Z_2$) return true
  else return false

L'algoritmo ha complessità ottimale $\mathbf{O}(n^2)$.

Per quanto riguarda la correttezza della risposta, osserviamo che se $V_2 \neq Z_2$ allora ovviamente $A \cdot B \neq B \cdot A$, quindi A e B non commutano.

Potrebbe però succedere che $V_2 = Z_2$ ma $A \cdot B \neq B \cdot A$: in tal caso l'algoritmo risponderebbe in modo errato.

È quindi necessario stimare la probabilità di errore P_e, che è la probabilità di estrarre X in $\{-1, 1\}^n$ tale che

$$X \cdot A \cdot B = X \cdot B \cdot A \text{ con } A \cdot B \neq B \cdot A.$$

In altri termini, posto $C = A \cdot B - B \cdot A$, risulta $X \cdot C = 0$ mentre $C \neq 0$.

Se $C \neq 0$ almeno una sua colonna $c = (c_1,.., c_n)$ non è nulla; in particolare deve esserci almeno un elemento diverso da 0. Possiamo supporre che sia c_1. Osserviamo che, se un vettore $X = (x_1, x_2,.., x_n)$ produce una segnalazione errata, cioè è tale che:

$$\Sigma_{k=1,n}\, x_k c_k = c_1 x_1 + \Sigma_{k=2,n}\, x_k c_k = 0$$

prendendo $X' = (-x_1, x_2,.., x_n)$, si ha:

$$\Sigma_{k=1,n}\, x'_k c_k = -c_1 x_1 + \Sigma_{k=2,n}\, x_k c_k \neq 0$$

Quindi per ogni vettore X che potrebbe fornire una segnalazione errata ce n'è almeno uno che fornisce la segnalazione giusta.

Ciò significa che per almeno la metà dei vettori x in $\{-1, 1\}^n$ deve valere che $\Sigma_{k=1,n}\, x_k c_k \neq 0$, pertanto la probabilità d'errore P_e è al più ½.

Non-commutatività $\in$ RP

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***", a patto di accettare un ragionevole livello di rischio, 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 può essere risolto da un algoritmo randomizzato che lavora in tempo polinomiale.

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 f: $\Sigma^* \rightarrow \Sigma^*$ è detta **funzione one-way** se:

- f può essere calcolata da un algoritmo in tempo polinomiale
- la sua inversa f^{-1} non può essere calcolata da alcun algoritmo randomizzato in tempo polinomiale nel seguente senso: per ogni algoritmo randomizzato Alg in tempo polinomiale, per n sufficientemente grande e per parole w in Σ^n selezionate a caso, la probabilità che sia w = Alg(f(w)) è al più 1/4.

Questa nozione, derivata in termini di complessità computazionale, ha trovato applicazioni in varie aree, dalla crittografia al commercio elettronico

L'esistenza di funzioni one-way è un difficile problema teorico che, come il 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 e 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 e 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.

Algoritmi di tipo **Montecarlo**

L'algoritmo di Miller-Rabin è un esempio di algoritmo probabilistico, più precisamente di algoritmo di tipo **Montecarlo**, cioè:

- Fornisce sempre risposta
- La risposta può essere errata
- La probabilità di errore è nota

"Risposte probabilmente vere in tempi certamente brevi"

La probabilità di una risposta corretta aumenta con il tempo a disposizione. Lo svantaggio principale è che in generale non è possibile decidere in modo efficiente se la risposta data è corretta.

Algoritmi di tipo **Las Vegas**

- Possono non rispondere
- L'eventuale risposta è sempre corretta
- È valutabile la probabilità di non risposta

"Risposte certamente vere in tempi probabilmente brevi"

Non vi è nessun upper bound sul tempo che può essere richiesto per ottenere una soluzione, anche se il tempo atteso richiesto per ogni istanza può essere piccolo e la probabilità di incontrare un'esecuzione che richiede tempo eccessivo è trascurabile.
La probabilità di fallimento su una istanza può essere resa arbitrariamente piccola ripetendo abbastanza volte lo stesso algoritmo sull'istanza .

La caratteristica distintiva è che rischiano di fare una scelta sbagliata che rende impossibile trovare la soluzione: la decisione sbagliata porta a un impasse.

Questi algoritmi pertanto ritornano una soluzione corretta o ammettono che le decisioni prese hanno portato all'impossibilità di trovare una soluzione.
In quest'ultimo caso basta risottomettere la stessa istanza allo stesso algoritmo per avere una seconda chance, indipendente dalla prima, di arrivare alla soluzione.

Gli algoritmi Las Vegas ritornano di solito un parametro *successo* che assume il valore "vero" se la soluzione è stata ottenuta, "falso" altrimenti.

Esempio: 8-regine

Algoritmo di backtracking

QEENS (k, *col*, *dd*, *dup*)

```
if (k > 8) stampa try
           return true
for j ← 1 to 8
      if (j ∉ col and j-k ∉ dd and j+k ∉ dup)
          try[k] ← j
          if QEENS (k+1, col∪{j}, dd∪{j-k}, dup∪{j+k})
             return true
return false
```

Chiamata principale: QUEENS(1, Φ, Φ, Φ)

La tecnica del backtracking implica di visitare sistematicamente i nodi dell'albero formato dai vettori k-promettenti, che possono cioè portare a una soluzione dopo aver sistemato la prime k regine.
In tal modo si ottiene la soluzione dopo aver esaminato 114 dei 2.057 nodi dell'albero.

Non è male, ma l'algoritmo non tiene conto di un fattore importante: nella maggior parte delle soluzioni non vi è nulla di sistematico nelle posizioni delle regine, al contrario le regine sembrano avere posizioni "casuali".

Questa osservazione suggerisce un algoritmo greedy Las Vegas che posiziona le regine a caso sulle righe successive, verificando che le regine sulla scacchiera non siano sotto scacco.

```
QEENS-LV ()
  col ← dd ← dup ← Φ
  k ← 1
  repeat nb ← Φ
       for i ← 1 to 8
            if (i ∉ col and i-k ∉ dd and i+k ∉ dup)
                nb ← nb ∪ { i }
        if (nb ≠ Φ
            j ← random (nb)
            try[k] ← j
            col ← col ∪ {j}
            dd ← dd ∪ {j-k}
            dup ← dup ∪ {j+k}
            k ← k+1
  until nb = Φ or k = 8
    return   (nb = Φ)  \\ in questo caso l'algoritmo ha successo
```

L'algoritmo Las Vegas è troppo disfattista: quando fallisce ricomincia tutte le volte dall'inizio.
D'altra parte l'algoritmo di backtracking cerca sistematicamente una soluzione che non ha nulla di sistematico.

Una combinazione dei due algoritmi pone un certo numero di regine sulla scacchiera in modo random e poi usa il backtracking per aggiungere le restanti regine senza riconsiderare la posizione delle prime poste a caso.

stopVegas, 1<= *stopVegas* <= 8, indichi quante regine devono essere sistemate random prima di iniziare la fase di backtracking.

```
QEENS-LV-MIX ()
  col ← dd ← dup ← Φ
  k ← 1
  repeat nb ← Φ
      for i ← 1 to 8
          if (i ∉ col and i-k ∉ dd and i+k ∉ dup)
              nb ← nb ∪ { i }
          if (nb ≠ Φ
              j ← random (nb)
              try[k] ← j
              col ← col ∪ {j}
              dd ← dd ∪ {j-k}
              dup ← dup ∪ {j+k}
              k ← k+1
      until nb = Φ or k > stopVegas
      if (nb ≠ Φ) return QEENS (k, col, dd, dup)
      else return false
```

dove *QEENS* (k, *col*, *dd*, *dup*) è la procedura di backtrack che parte dalla posizione "try" delle regine.

stopVegas	p	s	e	t
0	1,0	114,00	-	114,00
1	1,0	39,63	-	39,63
2	0,8750	22,53	39,67	28,20
3	0,4931	13,48	15,10	29,01
4	0,2618	10,31	8,79	35,10
5	0,1624	9,33	7,29	46,92
6	0,1357	9,05	6,98	53,50
7	0,1293	9,00	6,97	55,93
8	0,1293	9,00	6,97	55,93

p: probabilità di successo
e: numero atteso di nodi esplorati in caso di fallimento
s: numero atteso di nodi esplorati in caso di successo
$t = s + (1-p)\cdot e/p$: numero atteso di nodi esplorati se l'algoritmo è ripetuto finché si trova una soluzione

Algoritmi di tipo **Sherwood**

- Forniscono sempre una risposta
- La risposta è sempre corretta

Sono usati quando gli algoritmi deterministici noti per risolvere il problema vengono eseguiti molto più velocemente nel caso medio che nel caso peggiore.

Un algoritmo **Sherwood** non è più veloce in media dell'algoritmo deterministico da cui proviene.
Incorporare un elemento casuale permette all'algoritmo di ridurre, e qualche volta eliminare, la differenza tra buone e cattive istanze.

Algoritmi di tipo **Sherwood**

Esempio: randomized Quicksort

Randomized_partition (A, p, r)
    i ← random (p, r)
    scambia A[r] con A[i]
    return *Partition* (A, p, r)

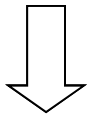
Randomized_quicksort (A, p, r)
    if (p < r)
        q ← *Randomized_partition* (A, p, r)
        Randomized_quicksort (A, p, q -1)
        Randomized_quicksort (A, q +1, r)

FINE

Come esistono algoritmi randomizzati che calcolano soluzioni esatte, esistono algoritmi randomizzati che calcolano soluzioni approssimate.

Un algoritmo randomizzato per un problema ha un rapporto di approssimazione $\rho(n)$ se, per un input qualsiasi di dimensione n, il **costo atteso** C della soluzione prodotta dall'algoritmo randomizzato è uguale al costo C* di una soluzione ottima a meno di un fattore $\rho(n)$.

$$\max\left[\frac{C(S)}{C(S^*)}, \frac{C(S^*)}{C(S)}\right] <= \rho(n)$$



Algoritmo randomizzato $\rho(n)$-approssimato

Una particolare istanza del problema 3-CNF può essere o no soddisfattibile e per essere soddisfattibile deve esistere un'assegnazione di valori per le variabili tale che ogni clausola risulti vera (uguale a 1).
Se un'istanza non è soddisfattibile, si può calcolare quanto sia "***vicina***" ad esserlo, cioè possiamo calcolare un'assegnazione di valori per le variabili che soddisfa il maggior numero possibile di clausole.

Il problema di massimizzazione così ottenuto è detto "**soddisfattibilità MAX-3-CNF**":

Data una formula in forma normale 3-congiuntiva, restituire un'assegnazione di valori per le variabili che massimizza il numero di clausole che risultano vere.

Ipotesi: nella formula ogni clausola deve essere formata da 3 letterali distinti e nessuna clausola deve contenere una variabile e la sua negazione.

Vogliamo dimostrare che assegnare casualmente a ciascuna variabile il valore 1 con probabilità ½ e il valore 0 con probabilità ½ è un algoritmo randomizzato 8/7-approssimato.

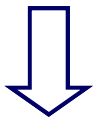
Teorema

Data una formula in forma normale 3-congiuntiva, con n variabili $x_1, x_2, \ldots, x_n$ ed m clausole, l'algoritmo randomizzato che assegna indipendentemente a ciascuna variabile il valore 1 con probabilità ½ e il valore 0 con probabilità ½ è un algoritmo randomizzato 8/7-approssimato.

Dimostrazione

Per i = 1, 2, …, m, definiamo la variabile casuale indicatrice:
Y_i = 1 {la clausola i è soddisfatta}

Se Y_i = 1 almeno uno dei letterali nella i-esima clausola vale 1. Poiché nessun letterale appare più di una volta nella stessa clausola e poiché una variabile e la sua negazione non possono apparire nella stessa clausola, i valori dei tre letterali in ogni clausola sono indipendenti.



Una clausola non è soddisfatta soltanto se tutti e tre i suoi letterali sono a 0 e quindi la probabilità che la clausola i non sia soddisfatta è $(½)^3$ = 1/8 e, come conseguenza, la probabilità che sia soddisfatta è 1- 1/8 = 7/8.

Pertanto $E[Y_i] = 7/8$.

Indicando con Y il numero totale di clausole soddisfatte, cioè:

$$Y = Y_1 + Y_2 + \ldots + Y_m$$

si ha:

$$E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] =$$

$$= \sum_{i=1}^{m} E[Y_i] =$$

$$= \sum_{i=1}^{m} 7/8 = 7m/8$$

m è un limite superiore per un numero di clausole soddisfatte, quindi il rapporto di approssimazione è al massimo $m/(7m/8) = 8/7$.

Algoritmi randomizzati

Algoritmi randomizzati

Un algoritmo probabilistico calcola una funzione f(x) se, avendo come input x, fornisce in uscita f(x) con probabilità almeno ¾.

La definizione è ben data perché è impossibile avere in uscita due diversi valori entrambi con probabilità ¾. Inoltre nella simulazione probabilistica di un algoritmo non deterministico il tempo di calcolo è limitato dal tempo non deterministico, del tutto accettabile in molte applicazioni.

Osserviamo che l'esecuzione di un algoritmo probabilistico che calcola la funzione f(x) non dà necessariamente f(x) sull'input x: l'algoritmo può pertanto commettere errore, ma la probabilità d'errore è limitata a ¼.
La probabilità d'errore è un parametro importante: quantifica il rischio di usare l'algoritmo e l'errore che siamo disposti ad accettare.

Sia ALG un algoritmo randomizzato che calcola la funzione f(x) con probabilità d'errore ¼. Un modo per ottenere un algoritmo che calcola la stessa funzione con probabilità d'errore inferiore a ¼ è quello di ripetere in maniera indipendente l'algoritmo un certo numero di volte sullo stesso input e dare in uscita la risposta che si presenta con maggiore frequenza.

```
for k ←1 to t
        y[k] ← ALG(x)
        z ← l'elemento più ripetuto in y[1], y[2], … y[t]
return (z)
```

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[t] sia inferiore a 1/2 . Poiché ci si aspetta che la frequenza di f(x) in y[1], …, y[t], si concentri intorno a ¾, la probabilità che la frequenza di f(x) in y[1], …, y[t] sia inferiore a ½ scende rapidamente all'aumentare di t.

Per ottenere una misura della probabilità di errore, osserviamo che la frequenza di f(x) in y[1], …, y[t] è una variabile aleatoria F distribuita come una binomiale B(t,p), per cui vale la disuguaglianza di Chernoff:

$$\text{probabilità}(F/t <= x) <= e^{-2t(x-p)^2}$$

$$x = 1/2,\ p = 3/4$$

$$\text{probabilità}(F/t <= 1/2) <= e^{-t/8}$$

Se siamo disponibili ad accettare un rischio d'errore d, basta ripetere l'algoritmo Alg t volte con t: $d = e^{-t/8}$, cioè $t = 8 \cdot \ln(1/d)$