Ancora le Regine

Il problema della disposizione di n regine su una scacchiera non viene affrontato con l'ausilio di Mathematica. *I programmi presentati non hanno nessuna pretesa di efficienza ma fanno risaltare la facilità di un approccio esplorativo a problemi non numerici che coinvolgono argomenti profondi di matematica computazionale.*

di Francesco Romani

Introduzione

Accusatemi di plagio, ma ogni volta che leggo gli articoli di Giustozzi & C. sui giochi matematici non riesco a resistere alla tentazione di provare a fare variazioni sul tema usando *Mathematica*. Figuratevi questa volta che ho trovato contemporaneamente ben due articoli sul problema delle n regine, uno su MCmicrocomputer (n.131) e uno sul *The Mathematica Journal* (piccolo il mondo, vero?). Reggetevi forte che cominciamo.

Il problema preso in esame è quello di disporre n regine su una scacchiera n x n in modo che nessuna di queste ne stia attaccando un'altra. Dal punto di vista matematico una prima formulazione è quella di trovare una matrice di 0 e 1 tale che non ci sono due 1 sulla stessa riga, colonna o diagonale.

Regine, Torri e permutazioni

Se invece delle regine lavorassimo con le torri la condizione sarebbe di trovare una matrice di 0 e di 1 tale che non ci sono due 1 sulla stessa riga, o colonna. L'insieme di tutte le soluzioni a questo problema è dato dalle matrici di permutazione ovvero dalle matrici ottenibili dalla matrice identica di dimensione n applicando una permutazione alle righe. In altre parole la permutazione ci dice, riga per riga, in quale colonna c'è l'unico 1. Siccome le permutazioni di n elementi sono n! allora le matrici di permutazione sono n! e tante sono, esattamente, le soluzioni al problema delle torri. Il problema delle regine ha come insieme di soluzioni un sottoinsieme delle matrici di permutazione con la proprietà che non ci sono due o più 1 sulla stessa diagonale. Gli algoritmi che presentiamo nel seguito lavorano su permutazioni dei primi n numeri interi, utilizzando le matrici di 0 e 1 solo per disegnare i grafici. La funzione **DQmat** stampa una matrice e **DQperm** trasforma una permutazione in una matrice e chiama **DQmat** per il plottaggio.

In[1]:=
```
DQmat[m_]:=
  ListDensityPlot[1 - Reverse[m],
                  FrameTicks -> None];
```
In[2]:=
```
DQperm[p_]:=DQmat[
    IdentityMatrix[Length[p]][[p]]];
```
In[3]:=
```
DQperm[{2, 5, 3, 1, 4}];
```

Vedi Figura 1.

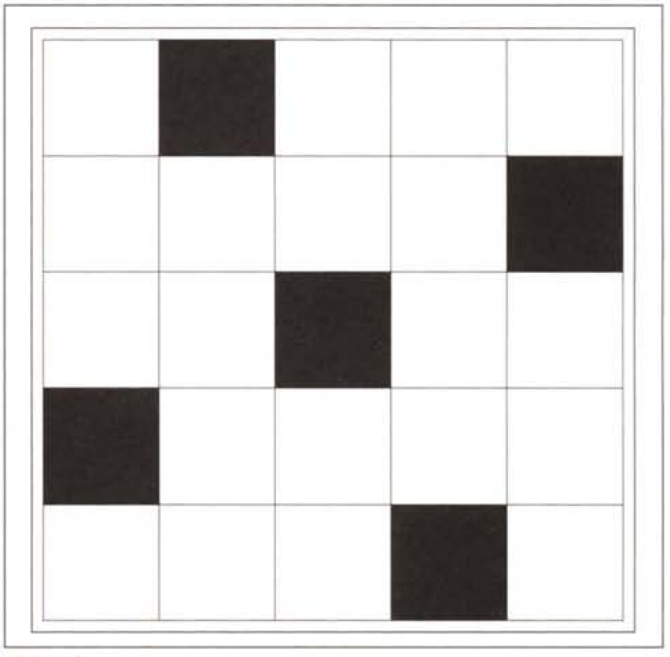


Figura 1.

Algoritmo 1: generazione diretta di tutte le soluzioni

Per trovare in un colpo solo tutte le soluzioni al problema delle regine basta generare le n! permutazioni e selezionare quelle che non hanno conflitti sulle diagonali. Se **x** è la permutazione in esame ed **r** è la permutazione identica (1,2,...,n) il test da superare si scrive

```
Map[Length,{Union[x+r],Union[x-r]}]
      == {n,n}
```

In altre parole si richiede che gli insiemi **x+r** e **x-r** non devono avere meno di n elementi. Il programma completo (dovuto a Ilan Vardi) è il seguente

In[4]:=
```
Reginel[n_] := Module[{r,p},
  r=Range[n];
  p=Permutations[r];
  Select[p, (Map[Length,
    {Union[#+r],Union[#-r]}]=={n,n})&]]
```

In[5]:=
```
s5=Reginel[5]
```
Out[5]=
```
{{1, 3, 5, 2, 4}, {1, 4, 2, 5, 3},
 {2, 4, 1, 3, 5}, {2, 5, 3, 1, 4},
 {3, 1, 4, 2, 5}, {3, 5, 2, 4, 1},
 {4, 1, 3, 5, 2}, {4, 2, 5, 3, 1},
 {5, 2, 4, 1, 3}, {5, 3, 1, 4, 2}}
```
Vediamo i tempi (in secondi) e il numero di soluzioni trovate per vari valori di n. La macchina usata è un Centris 650 (68040 a 25 MHz, 24 Megabyte di RAM), con *Mathematica* 2.2.1 Enhanced con 3.5 Mb per il FrontEnd e 16 Mb per il Kernel)

n	t	sol
5	1.46	10
6	9.21	4
7	71.33	40
8	631.88	92

La complessità è chiaramente superiore a n!. Ad una analisi accurata, considerando il costo degli ordinamenti necessari per fare l'unione, risulta O(n! n log n). Questo algoritmo così elegante (posso dirlo tanto non è mio!) non ha quindi un futuro computazionale.

Algoritmo 2

Per avere qualche speranza di arrivare a trovare tutte le soluzioni è necessario scartare roba mentre si fa la generazione, evitando così di lavorare su n! oggetti. Una prima semplificazione è quella di partire con una regina nella quarta cella della prima riga (utilizzando il fatto noto che c'è sempre una soluzione con una regina a quattro celle da un angolo). Questo accorgimento permette di avere gia posizionato in modo corretto un elemento e di ridurre l'insieme delle possibili soluzioni.
Una volta messa la prima regina, si considera l'insieme di tutte le coppie che rappresentano soluzioni ammissibili per le prime due righe. A partire da questo insieme si generano le soluzioni ammissibili su tre righe, ecc. Il trucco per fare le cose in modo efficiente è quello di usare quanto più possibile le funzioni interne di *Mathematica*. Sia **r4** la lista {1,2,3,5,...,n}. La funzione **Outer[JJ,x,r4]** applica la funzione **JJ** a tutte le coppie di elementi di **x** e **r4**. La funzione **JJ** è costruita in modo da appendere il secondo argomento in coda al primo e poi viene sostituita da **List**. Il risultato di **Increase** è di aggiungere rapidamente ad un insieme di soluzioni possibili su k righe tutte le combinazioni di regine sulla k+1-esima riga (la colonna 4 resta vuota perché già occupata in prima riga).
In[1]:=
```
Increase[x_List]:=
   Flatten[Outer[JJ,x,r4]]/.JJ->List;
```
In[2]:=
```
JJ[JJ[a__], b_] := JJ[a, b]
```
La funzione **Sel** filtra un insieme di soluzioni possibili su m righe eliminando quelle che presentano ripetizioni su una colonna o su una diagonale.
In[3]:=
```
sel[x_List,m_]:=Select[x,
  Map[Length,{Union[#],Union[#+r],
              Union[#-r]}]=={m,m,m}&]
```
Il programma **QS** chiama n-1 volte **Increase** e **Sel** generando tutte le soluzioni che cominciano per 4.
In[4]:=
```
QS[n_]:=Module[{rt,q},
  rt=Range[n];
  r4=Complement[rt,{4}];
  q={JJ[4]};
  Do[ q=Increase[q];
      r=Range[m];
      q=Apply[JJ,sel[q,m],{1}],
     {m,2,n}];
  q/.JJ->List];
```
In[5]:=
```
QS[6]
```
Out[5]=
```
{{4, 1, 5, 2, 6, 3}}
```
Vediamo i tempi (in secondi) e il numero di soluzioni trovate:

n	t	sol
6	1.76	1
7	6.83	6
8	29.98	18
9	129.28	18
10	594.7	93

Algoritmo 3

Un inconveniente dell'algoritmo di cui sopra è che si continua a provare soluzioni in tutte le colonne, tranne la 4, anche se altre colonne sono piene. Si può aumentare l'efficienza spezzando la computazione in due parti: si agisce come sopra ma solo per n/2 righe e poi si prova a completare, una per una, tutte le soluzioni parziali trovate.
La funzione **QS1** è come **QS**, ma si ferma dopo ⌈n/2⌉ passi, **QS[s,n]** parte da una soluzione **s** su m righe e arriva fino in fondo. Il programma di prova si ferma appena trovata **almeno una** soluzione.
In[1]:=
```
QS[s_,n_]:=Module[{rt,q},
  rt=Range[n];
  r4=Complement[rt,s];
  q={JJ@@s};
  Do[ q=Increase[q];
      r=Range[m];
      q=Apply[JJ,sel[q,m],{1}],
              {m,Length[s]+1,n}];
  q/.JJ->List];
```
In[2]:=
```
t[n_]:=Timing[
     u=QS1[n];
     r=Scan[If[(r=QS[#,n])=!={},
                 Return[r]]&,u]];
```
Vediamo i tempi in secondi che questo algoritmo impiega a trovare una soluzione.

n	t
6	0.76
7	1.61
8	3.96
9	13.55
10	36.9
11	166.05
12	397.0

Rinunciando a trovare tutte le soluzioni si è guadagnato un fattore 10 in efficienza, tuttavia l'elaborazione di **t[13]** si interrompe dopo avere esaurito i 16 Mb (16 megabyte!!!) di RAM assegnati al Kernel.

Algoritmo 4

Per trovare una soluzione per n=13 ho provato a dividere la computazione in tre parti. La funzione **QS1[n,k]** si ferma dopo k passi, **QS2[s,n,k]** parte da s e fa k passi.

In[1]:=
```
u=QS1[13,4];
Length[u]
```
Out[27]=
```
454
```
Si generano tutte le 454 soluzioni parziali su 4 righe.
In[2]:=
```
Timing[v=
  Union[Join@@Map[QS2[#,13,8]&,u]];]
```
Out[27]=
```
{6553.48 Second, Null}
```
In[28]:=
```
Length[v]
```
Out[28]=
```
54766
```
In meno di due ore si generano tutte le 54766 soluzioni parziali su 8 righe.
In[29]:=
```
Timing[s13=Scan[If[(r=QS[#,13])=!={},
      Return[r]]&,v]]
```
Out[29]=
```
{19.3667 Second,
 {{4,1,3,5,12,8,11,13,2,6,9,7,10}}}
```
E in meno di 19 secondi si trova una soluzione per n=13.

Algoritmo 5: ricerca euristica

Per risparmiare tempo e memoria durante la ricerca di una soluzione con n elevato si può mettere insieme un programma di ricerca euristica. Si lavora su un insieme di soluzioni parziali di varie lunghezze, espandendo un elemento per volta e aggiungendo il risultato all'insieme, fino a che non si è trovata una soluzione oppure l'insieme è vuoto. Questo metodo è usato, con infinite varianti, in tutti i casi in cui si cerca la soluzione in uno spazio di stati connesso ad albero (come nel nostro caso) o in generale in un grafo (in giochi come gli scacchi si può ritornare ad una posizione precedente). La chiave di volta del metodo è la strategia di scelta dell'elemento da espandere. Le strategie più brutali dei metodi ad esaurimento sono quelle di espandere i nodi più antichi (ricerca **Breadth First**, ovvero l'insieme di nodi è strutturato a coda) o quelli più recenti (ricerca **Depth First**, ovvero l'insieme di nodi è strutturato a pila. N.B. se lo spazio di ricerca è infinito la ricerca può non terminare). Una strategia più raffinata consiste nell'usare una coda a priorità ed espandere gli elementi in base ad una funzione di valutazione che sceglie i più promettenti.
Per il nostro problema iniziamo con una ricerca **Breadth First** realizzando le funzioni di gestione di una coda. **`InQ[Q,x]`** inserisce **`x`** nella coda **`Q`**, **`OutQ[Q]`** estrae un elemento aggiornando la coda.
In[1]:=
```
InQ[Q_,x_]:=Join[Q,x]
OutQ[Q_]:={First[Q],Rest[Q]}/;Q=!={}
```
Poiché si tratta un elemento per volta, si aggiungono elementi non presenti nelle righe precedenti e il controllo si semplifica.
In[2]:=
```
sel1[x_List,m_]:=Select[x,
  Map[Length,{Union[#+r],
              Union[#-r]}]=={m,m}&]
```
La funzione **`process`** estrae un elemento dalla coda, lo espande e rimette in coda gli eventuali elementi risultanti, restituendo **`False`** se l'insieme **`q`** contiene almeno una soluzione

In[3]:=
```
process[n_]:=Module[{item,new},
  {item,Q}=OutQ[Q];
  m=Length[item]+1;
  r=Range[m];
  r4=Complement[rt,item];
  q=sel1[Map[Append[item,#]&,r4],m];
  Q=InQ[Q,q];
  (m<n)||(q=={})]
```
Il programma di ricerca cicla finché **`process`** continua a valere **`True`**.
In[4]:=
```
t[n_]:= (rt=Range[n];
         Q={{4}};
         While[process[n],];
         First[q]);
```
Vediamo i tempi in secondi richiesti da questo algoritmo per trovare una soluzione.

n	t
6	0.9
8	13.31

Le cose non vanno molto bene! Fidando nel fatto che le soluzione sono tante, e andando in profondità, si dovrebbe fare presto a trovarne qualcuna, proviamo a cambiare il tipo di ricerca e passare alla **Depth First**.
In[5]:=
```
InQ[Q_,x_]:=Join[x,Q]
```
(Come vedete non serve un grande sforzo programmativo per implementare una pila o una coda in *Mathematica*.)
Vediamo i tempi, in secondi, richiesti dalla nuova versione.

n	t
8	1.56
9	1.33
13	22.41
14	547.90
15	64.23
16	916.48
17	992.31
18	4964.54
19	2388.73

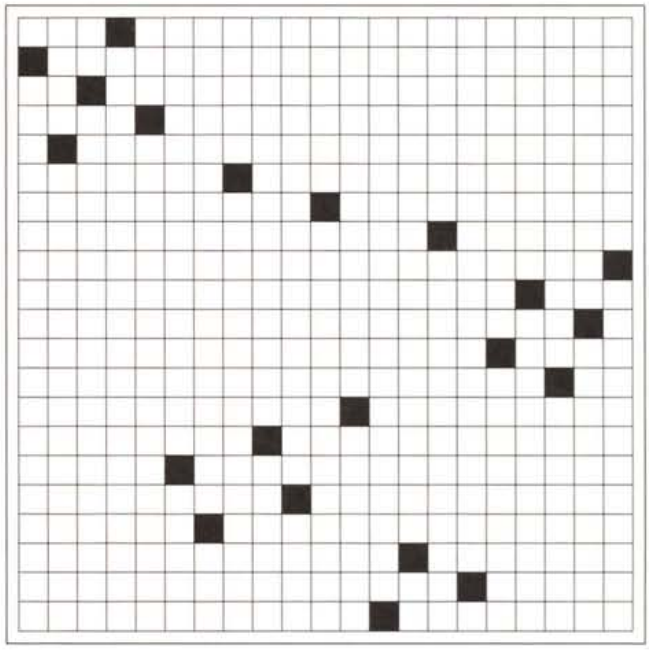


Figura 2 ▶

Bibliografia

J.Freeman, **A neural Network Solution to the n-Queens Problem.** The Mathematica Journal. Vol. n. 3. Summer 1993, pp.52-56.
C.Giustozzi, F.Balena. **Regine e Amazzoni, il grande ritorno.** MCmicrocomputer, n. 131, Luglio/Agosto 1993., pp. 206-210.
I. Vardi, **Computational Recreations in *Mathematica*.** (Addison-Wesley, 1991).
S. Wolfram, ***Mathematica*. A System for Doing Mathematics by Computer, II Edition**. (Addison Wesley, 1991).

20 29263.80
21 1734.37

Si nota come, stranamente, la ricerca di soluzioni di dimensione dispari sia più facile. La soluzione per n=21 è in Figura 2.

Generazione diretta di soluzioni con n grande

Confrontando la Figura 2 con la Figura 1 si vede che in alto a sinistra è riportata pari pari una soluzione per n=5, al centro a destra si intravede una soluzione per n=4. È facile capire che combinando soluzioni di dimensioni piccole si possono ottenere soluzioni grandi. Lasciando al lettore volenteroso il problema di trovare una soluzione di ordine n primo grande (per esempio1069), vi faccio vedere come trovare soluzione grandi attraverso il **prodotto tensoriale** di soluzioni piccole. Il prodotto tensoriale di due matrici di dimensioni rispettivamente h×h e k×k è una matrice di dimensioni hk×hk ottenuta incollando copie della seconda matrice moltiplicate per gli elementi della prima. In particolare il prodotto tensoriale di matrici di permutazione è ancora una matrice di permutazione. Vediamo in pratica come si costruisce una soluzione di ordine 25 a partire da una soluzione di ordine 5.
Facciamo prima un programma che verifica se una permutazione è una soluzione del problema delle regine e applichiamolo alla soluzione di figura 1.

In[1]:=
```
Verify[s_] := Module[{n},
  n=Length[s];
  r=Range[n];
  (Map[Length,
    {Union[s],Union[s+r],Union[s-r]}]==
     {n,n,n})]
```
In[2]:=
```
Verify[ss=Reginel[5][[4]]]
```
Out[2]=
```
True
```
Per fare il prodotto tensoriale si potrebbe passare dai vettori alle matrici di permutazione ma è più semplice applicare **Outer** con una semplice funzione di riordinamento (nel seguito scritta per n=5).

In[3]:=
```
f[x_,y_]:=y+5(x-1);
s2=Flatten[Outer[f,ss,ss]]
```
Out[3]=
```
{7, 10, 8, 6, 9, 22, 25, 23, 21, 24, 12,
  15, 13, 11, 14, 2, 5, 3, 1, 4, 17, 20,
  18, 16, 19}
```
In[4]:=
```
Verify[s2]
```
Out[4]=
```
True
```
s2 è una soluzione di ordine 25 *(Figura 3)*.
Infine, in meno di tre secondi generiamo (senza stamparla o disegnarla) una soluzione di dimensione 625.

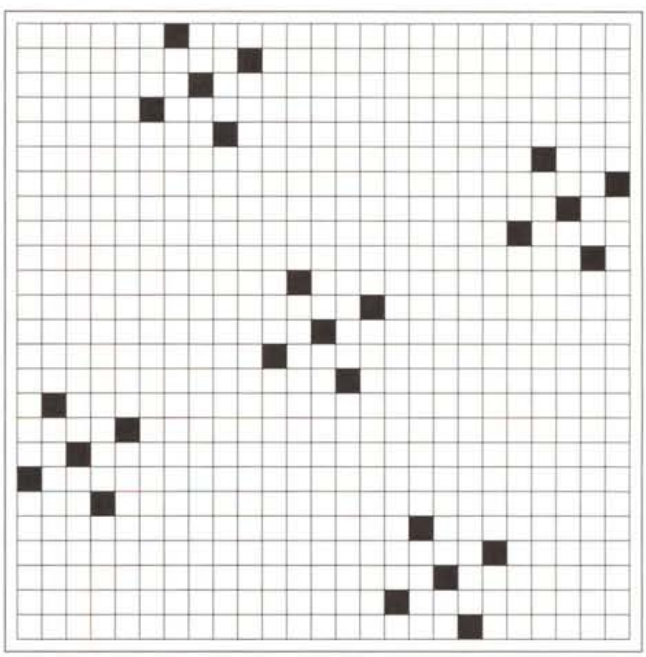


Figura 3

In[5]:=
```
f[x_,y_]:=y+25(x-1);
Timing[s4=Flatten[Outer[f,s2,s2]];
      Verify[s4]]
```
Out[5]=
```
{2.45 Second, True}
```

Soluzioni attraverso le reti neuronali

Per finire, ricordo ai lettori interessati che sul numero estivo del *The Mathematica Journal* c'è un articolo di J.Freeman sulla soluzione del problema delle n regine attraverso una rete neuronale. La rete viene impostata in modo da associare al problema un sistema fisico in cui alle soluzioni corrispondono dei minimi di energia. La risoluzione di un sistema di equazioni differenziali ordinarie permette di simulare il comportamento della rete e la sua convergenza a minimi locali che (spesso) rappresentano soluzioni. L'articolo richiede una buona dose di prerequisiti matematici ed è consigliabile ai lettori più esperti. Gustoso ma duro da digerire è anche il capitolo sul problema delle n regine nel libro di Ilan Vardi citato in Bibliografia.

MC

Francesco Romani è raggiungibile tramite Internet all'indirizzo romani@di.unipi.it

Angolo multimediale

Le schede tecnologicamente più avanzate
COMPATIBILI con 3 STANDARD OLTRE AL SOUND BLASTER PRO

STRATOS PRO	L. 197.000
AQUILA SOUND PRO 16 Bit	L. 299.000
KIT STRATOS+CD ROM MITSUMI	L. 609.000
KIT AQUILA+CD ROM MITSUMI	L. 720.000
CD ROM MITSUMI con CTRL	L. 390.000

Angolo telematico

distribuzione di 17 modelli tra modem e fax-modem esempi:

Modem interno	L. 52.000
Modem esterno	L. 93.000
Fax-modem interno	L. 93.000
Fax-modem esterno	L. 129.000
Pocket fax-modem	L. 178.000
14400 Fax-modem int. o esterno	L. 420.000

- Coprocessori
- RAM
- Schede di Rete
- Stampanti

prezzi particolari

Speciale Hard Disk:

HD 130MB	L. 280.000
HD 180MB	L. 339.000
HD 210MB	L. 374.000
HD 250MB	L. 428.000
HD 340MB	L. 562.000

HIDATA

PUNTI VENDITA IN ROMA:

HISHOP
Numidio Quadrato - Tuscolana
Via Scribonio Curione, 79 M
Tel. 06/76966900

HISHOP
Monte Mario - Trionfale
Via Andrea Angiulli, 6c
Tel. 06/3380095

PRINTEMPS
Prati
Via Angelo Emo, 99
Tel. 06/6374413

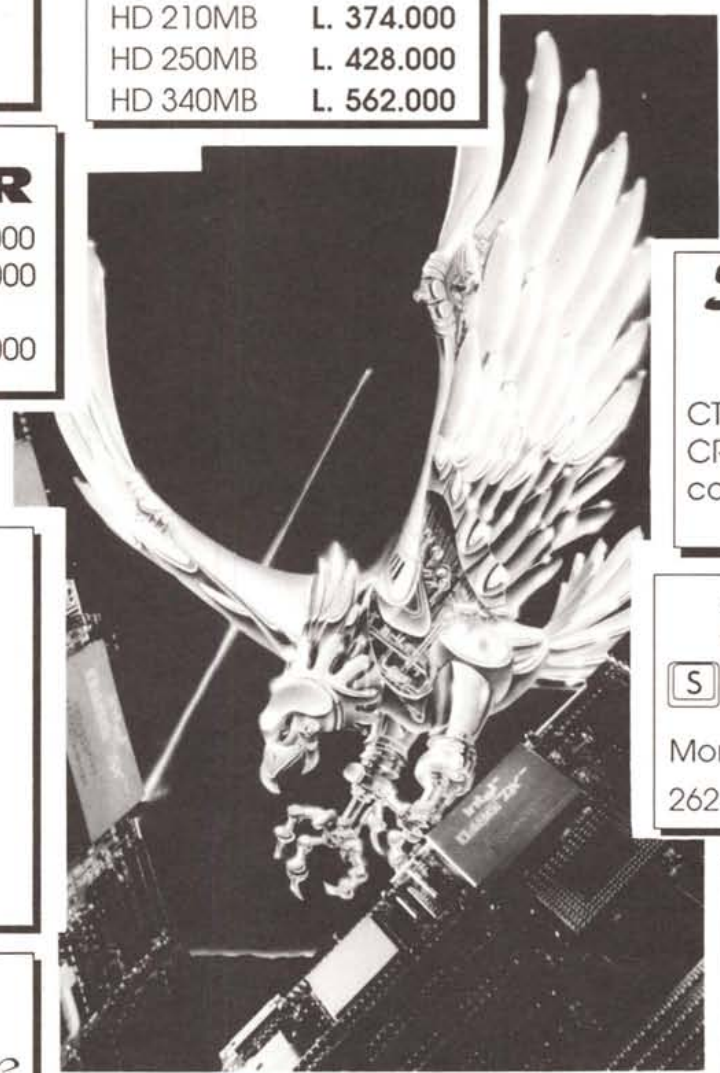
CENTRO DISTRIBUZIONE PER SOLI RIVENDITORI:
Roma
Tel. 06/3377224 - 3378848
Fax 06/3383650

Prezzi IVA esclusa

MONITOR

VGA Mono	L. 176.000
VGA Colore	L. 365.000
Multiscan colore	L. 428.000

HIDATA è un marchio registrato.



Speciale VESA:

CTRL INT. VESA **L. 75.000**
CRTL INT. VESA CACHE con microprocessore
L. 290.000

Dischetti

da 1,44 preformattati certificati
"free error"
Lit. 1.000 cad.

HANDY SCANNER

Mono	L. 189.000
262.000 colori	L. 540.000

Tavolette Grafiche

professionali con penna ottica

6x6	L. 189.000
12x12	L. 266.000
12X18	L. 554.000

FORTI SCONTI PER RIVENDITORI

GRUPPI DI CONTINUITÀ
"No break"
da Lit. 449.000
da 600 a 1500 VA

SCHEDE MADRI

486 40 MHZ DLC	L. 290.000
486 INTEL VESA 33	L. 679.000
486 INTEL VESA 50	L. 939.000
386 SX40	L. 139.000
386 DX40	L. 185.000