a cura di Francesco Romani

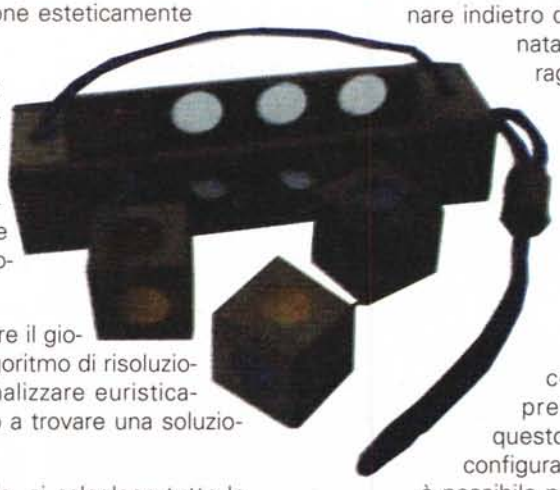
# Soluzione di problemi mediante ricerca in spazi di stati

**Continuando la serie dedicata alla "Enigmistica Computazionale" vediamo quello che forse è l'approccio classico alla risoluzione dei problemi complessi: la ricerca nello spazio delle possibili configurazioni guidata dalla conoscenza del gioco. Presentiamo un motore di ricerca generico e poi lo utilizziamo per trovare la soluzione di un semplice rompicapo basato su cubetti colorati.**

## Introduzione

Consideriamo un solitario che richiede di effettuare una serie di mosse (decise secondo le regole del gioco) fino ad arrivare ad una situazione particolare detta *goal*.

L'esempio più famoso di gioco di questo tipo è il cubo di Rubik in cui si parte da una configurazione disordinata e si cerca di ricostruire la configurazione con tutte le facce dello stesso colore (in alternativa si parte dalla configurazione di base e si cerca di ottenere un'altra configurazione esteticamente valida).



I giocatori "umani" spesso effettuano una serie disordinata di mosse apparentemente casuali e se il gioco è difficile (come il cubo) si stufano, rinunciano e lo mettono da parte. I giocatori più abili invece sanno come muovere e riescono in breve a trovare la soluzione.

Il nostro interesse consiste nel risolvere il gioco attraverso l'uso del computer. L'algoritmo di risoluzione che presentiamo consiste nell'analizzare euristicamente un albero di configurazioni fino a trovare una soluzione (o a dichiarare fallimento).

Si parte da una configurazione iniziale, si calcolano tutte le configurazioni raggiungibili da essa, per ognuna se ne dà una valutazione per poi inserirla in una coda tenuta ordinata secondo la "bontà". Quindi si prende la configurazione più promettente (quella in cima alla coda), si calcolano la mosse possibili a partire da questa, si mettono in coda le configurazioni risultanti, si prende la configurazione migliore e così via.

Facciamo alcune considerazioni:

- Questo algoritmo non rispecchia il modo di comportarsi di un giocatore umano. I giocatori umani in genere ragionano in termini di "mosse", e spesso non è possibile tornare indietro dopo avere effettuato una determinata mossa. Il nostro algoritmo invece ragiona in termini di "configurazioni" e può facilmente tornare indietro di molte mosse semplicemente scegliendo una configurazione messa da parte tempo prima. Vedremo in una puntata successiva l'algoritmo più complicato che simula il giocatore umano.
- È importante memorizzare le configurazioni già visitate e non prenderle più in considerazione, in questo modo ci si muove in un albero di configurazioni e non in un grafo (ovvero non è possibile percorrere un anello di configurazioni). L'assenza di cicli ci garantisce che se una soluzione

esiste questa potrà essere trovata in un tempo finito.

- L'efficienza dell'algoritmo dipende dalla funzione di valutazione e dalla natura del problema, nel caso migliore la soluzione viene trovata subito, nel caso peggiore l'algoritmo si riduce alla visita esaustiva di tutte le configurazioni.
- Anche se l'algoritmo ha sempre termine, le richieste di memoria e di tempo crescono in modo esponenziale e molti problemi sono intrattabili. Un programma di questo tipo per gli scacchi richiederebbe molta più memoria del numero di protoni dell'universo e un tempo di calcolo superiore a quello trascorso dal Big Bang.
- Algoritmi di questo tipo sono molto usati in informatica per risolvere in modo euristico problemi complessi.

# L'algoritmo di ricerca euristica

Vediamo il programma *Mathematica* che implementa l'algoritmo per un problema generico.

Dapprima realizziamo una coda a priorità. **InCoda[x,i]** inserisce in coda gli elementi di **x** che non sono ancora stati visitati, mettendo, nell'ordine, il valore della funzione, l'elemento e un numero **i** che rappresenta il livello (il numero di mosse dallo stato di partenza) di quella configurazione. L'uso di **Union** garantisce che gli elementi della coda sono tenuti in ordine senza ripetizioni.

*In[1]:=*
```
InCoda[x_List,i_Integer]:=Module[{v},
   v=Select[x,new];
   CODA = Union [CODA,
      Transpose[{fv/@v,(i+1)&/@v,v}]]]
```

**OutCoda** rende l'elemento in testa alla coda e la aggiorna, se la coda è vuota viene reso **Null**.

*In[2]:=*
```
OutCoda := Module [{f,l,s},
  If[Length[CODA] > 0,
    {f,l,s} = First[CODA];
    CODA = Rest[CODA];
    {f,l,s},
       {0,0,Null}]]
```

Il programma di inizializzazione assegna il valore **0** alla funzione di valutazione dello stato vuoto, definisce una funzione **new** che ricorda gli argomenti con cui è chiamata e per ognuno di essi rende **True** la prima volta e **False** le successive. Viene inizializzato lo stato e la coda.

*In[3]:=*
```
initialize:=(
  fv[Null]=0;
  Clear[new];
  new[x_]   :=(new[x]=False;True);
  STATO     = StatoIniziale;
  new[STATO];
  CODA      = {};
  InCoda[Mosse[STATO],1])
```

Il programma di ricerca effettua l'inizializzazione e continua a prelevare ed espandere lo stato in testa alla coda fino a che si trova un elemento con funzione di valutazione nulla (**successo**) oppure la coda è vuota (**fallimento**).

*In[4]:=*
```
go := (
  initialize;
  While[fv[STATO]>0,
    {f,l,STATO}=OutCoda;
    trace[{f,l}];
    If[f>0,InCoda[Mosse[STATO],l]]];
  If[STATO =!= Null,
      Print["Successo"];
      show[STATO],
      Print["Fallimento"]])
```

Per avere un programma funzionante sono da definire

- **StatoIniziale** : lo stato da cui si deve partire.
- **Mosse[stato]**: l'insieme degli stati raggiungibili da **stato**.
- **fv[stato]**: una funzione che calcola il valore di **stato**, **fv[stato]=0** significa che si è raggiunto l'obiettivo.
- **show[stato]**: una funzione che stampa o disegna il valore di stato.
- **trace[{f,l}]** : una funzione che stampa o disegna informazioni sul progresso dell'algoritmo. (N.B. **trace** può restare indefinita).

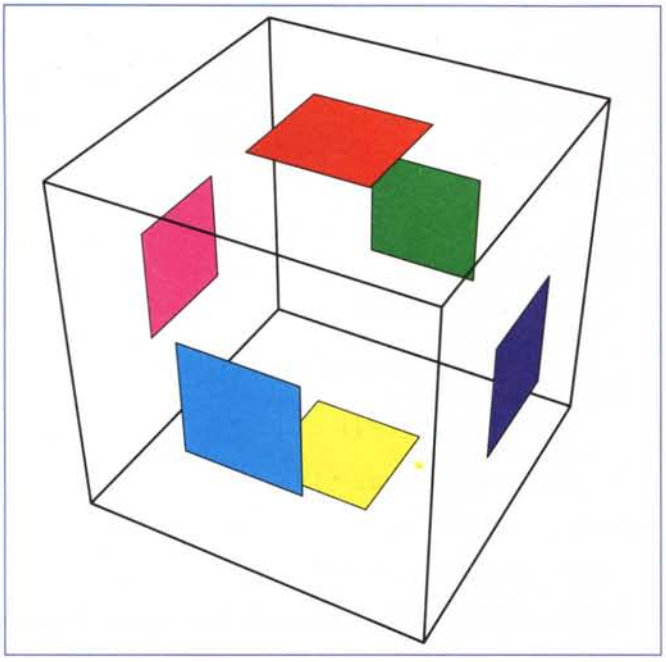
# Il problema dei cubetti

Il gioco che prendiamo ad esempio è un semplice rompicapo da tavolo formato da 4 dadi che hanno punti colorati sulle varie facce e un supporto di legno che permette di assemblare i 4 dadi uno accanto all'altro in modo che solo i 4 punti colorati laterali siano visibili. Il problema consiste nel disporre i dadi in modo che ognuna delle sequenze in vista sia formata da quattro colori diversi.

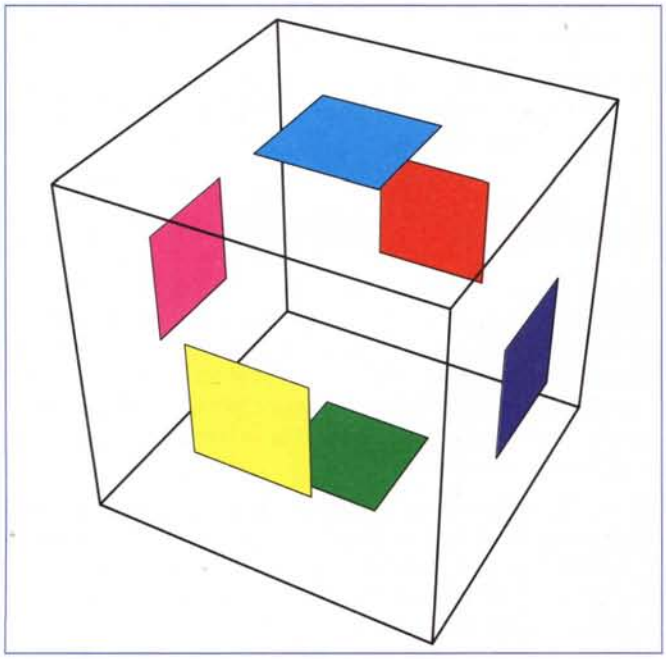
Non è evidente in base a quale legge siano stati assegnati i colori e quante soluzioni esistano. Il gioco, in legno, è stato comprato su una bancarella e non so neppure come si chiama; l'unica cosa certa è che mia moglie riesce sempre a risolverlo a mano ed io non ci riesco mai!

La prima decisione da prendere è il formato di rappresentazione interno. Decidiamo di rappresentare i colori con le lettere maiuscole iniziali del nome inglese.

*In[1]:=*
```
col[R]=Red;
col[G]=Green;
col[C]=Cyan;
col[Y]=Yellow;
col[M]=Magenta;
col[B]=Blue;
```



Le facce del dado sono ordinate in modo che alla lista **{R,G,Y,C,M,B}** corrisponde il dado di **Figura 1**.



Definiamo ora due tipi di mosse: la mossa **M1** ruota il dado in avanti di una posizione, il dado **{R,G,Y,C,M,B}** diviene quindi **{C,R,G,Y,M,B}**, (**Figura 2**).

La mossa **M2** ruota il dado di una posizione a destra tenendolo in piano avanti di una posizione, il dado **{R,G,Y,C,M,B}** diviene quindi **{R,M,Y,B,C,G}**, (**Figura 3**).

La funzione **show** mostra le 4 linee di colori come righe di pallini e i colori laterali, invisibili nel gioco reale come linee colorate. Il secondo parametro (opzionale) permette di mettere una intestazione al grafico.

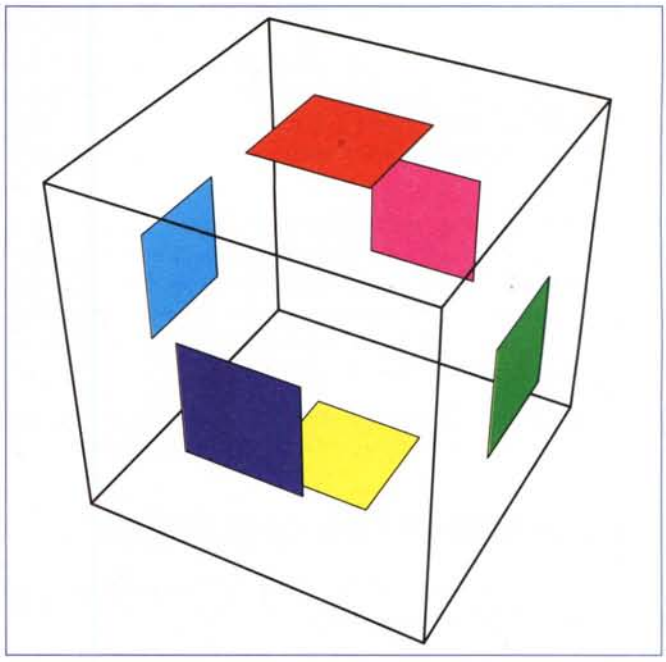


*Figura 3*

*In[2]:=*

```
show[mat_,opts___]:=Module[{arr,i,j},
  l=Length[mat];
  Show[Graphics[{
    Table[{Thickness[0.01],
           Line[{{j-0.4,0.2},{j+0.4,0.2}}],
           Line[{{j+0.4,4.2},{j-0.4,4.2}}],
           col[mat[[j,5]]],
           Line[{{j+0.4,0.2},{j+0.4,4.2}}],
           col[mat[[j,6]]],
           Line[{{j-0.4,4.2},{j-0.4,0.2}}]},
     {j,l}],
    PointSize[0.15],
    Table[{col[mat[[j,i]]],Point[{j,4.6-i}]},
       {i,l},{j,l}]}],
    AspectRatio->1,
    PlotRange->{{0.1,4.4},{0.1,4.4}},
    opts];];
```

Lo stato iniziale consiste in 4 cubetti tutti diversi, in una posizione prestabilita, ed è raffigurato in **Figura 4**.

*In[3]:=*

```
c1={R,C,Y,R,Y,Y};
c2={R,C,Y,R,G,G};
c3={R,C,C,Y,G,G};
c4={R,C,G,Y,G,G};
StatoIniziale = {c1,c2,c3,c4};
show[StatoIniziale,PlotLabel->"STATO INIZIALE"]
```

Vediamo quanti sono i possibili stati. Ai fini del gioco non serve cambiare l'ordine dei dadi e quello lo consideriamo quindi immutabile. Ogni dado può stare coricato su una tra 6 facce e

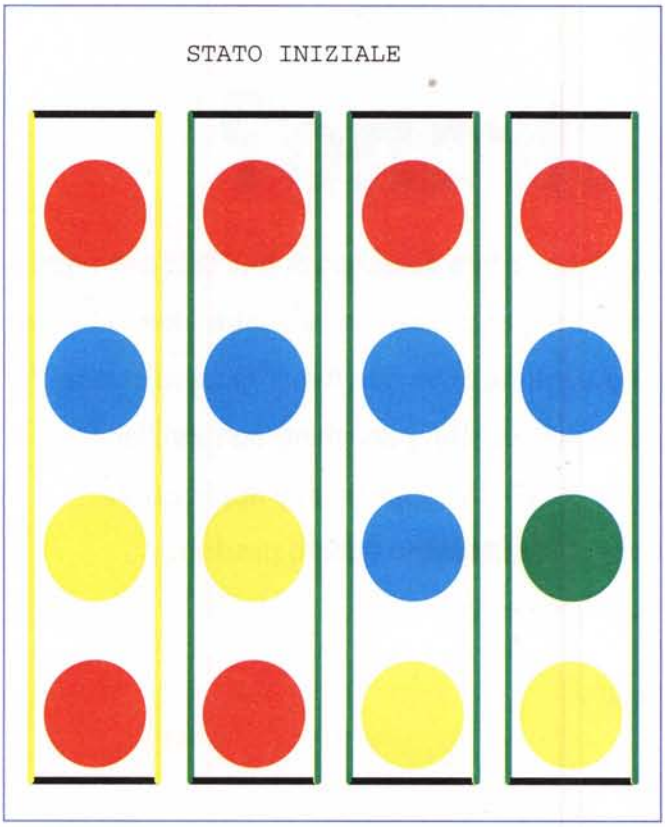



*Figura 4*

può essere orientato in 4 modi per cui vi sono $(6 \cdot 4)^4 = 331776$ configurazioni distinte.

Per muoversi attraverso tutto lo spazio degli stati bisogna trovare un insieme di mosse capace di mettere un qualsiasi dado in una qualsiasi posizione. È facile mostrare che per ogni dado è sufficiente realizzare le rotazioni di una posizione rispetto a due assi.

Le mosse vengono implementate come una permutazione. **Mosse[stato]** calcola tutte le configurazioni raggiungibili da **stato** con una sola mossa.

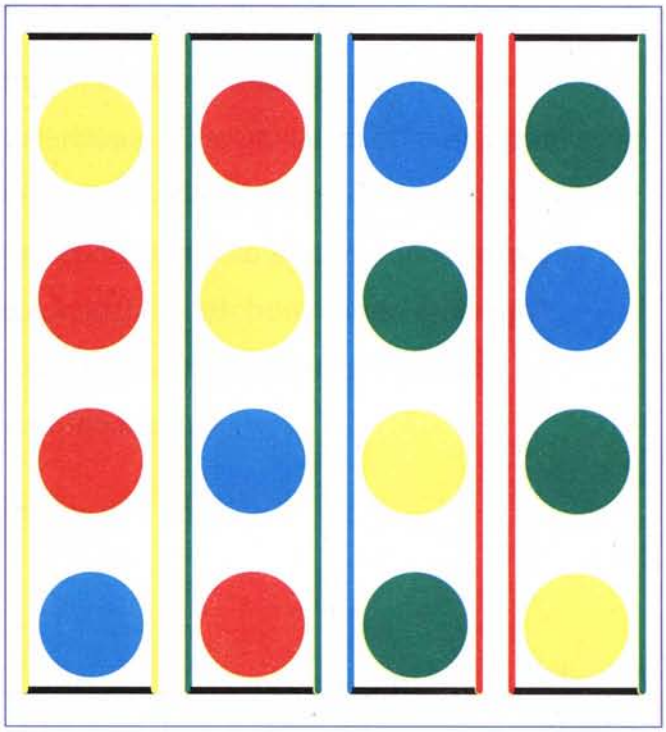
*In[4]:=*
```
M1[x_List]:=x[[{2,3,4,1,5,6}]];
M2[x_List]:=x[[{1,5,3,6,4,2}]];
M1[x_,i_]:=Module[{v},
  v=x; v[[i]]=M1[x[[i]]];v];
M2[x_,i_]:=Module[{v},
  v=x;[[i]]=M2[x[[i]]];v];
Mosse[stato_]:=Flatten[
   M1[stato,#],M2[stato,#]}&/@{1,2,3,4},1]
```

La funzione di valutazione è molto semplice e si limita a contare il numero di coppie di colori uguali sulla stessa riga.

*In[5]:=*
```
fv[stato_]:=Module[{s},
   s=Take[Transpose[stato],4];
   16-Plus@@(Length[Union[s[[#]]]]&/@{1,2,3,4})]
```

La funzione **trace** si limita a stampare il numero di elementi nella coda e il livello della configurazione sotto esame.

*In[6]:=*
```
trace[{f_,l_}]:=
    Print[l," ",Length[CODA]];
```



Ora possiamo far partire il motore di ricerca e ammirare la soluzione, **(Figura 5)**. Quando viene trovata la soluzione (ottenuta in 16 mosse) la coda conteneva 363 stati (tre ordini di grandezza sotto la dimensione dello spazio degli stati).

*In[7]:=*
```
go
```

*Out[7]=*
```
1 7
2 14
3 21
...
15 357
16 363
Successo
```

## Conclusioni

Il nostro scopo è raggiunto! Guardando la figura 5 è possibile mettere i dadi reali nella posizione giusta e dire alla moglie "ci sono riuscito anch'io".

Con i dadi in mano si nota anche che esistono due possibili posizioni per i dadi 3 e 4 e quindi vi sono almeno 4 soluzioni. In una prossima puntata cercheremo di simulare il comportamento di un giocatore umano.