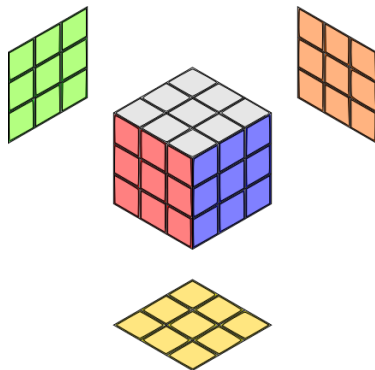


Il cubo di Rubik (e come risolverlo)

Stefano Angeleri, Alessandro Menti, Mattia Zago

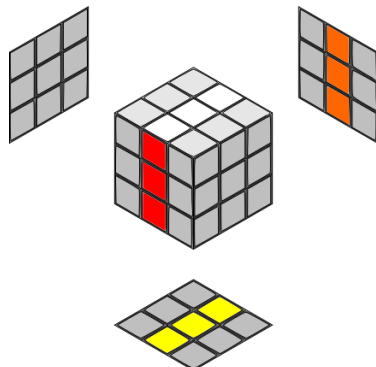
Alcune definizioni

- ▶ Considereremo un cubo 3×3
- ▶ Ogni faccia (*side*) ha un colore standard a essa associato (vedi figura)
- ▶ Ognuno dei nove pezzi di ogni faccia è detto *facelet*
- ▶ Il cubo ha 3 colonne/righe (*columns/rows*), 3 colonne laterali (*lateral columns*), 4 angoli (*corners*) e 8 spigoli (*edges*)



Alcune definizioni

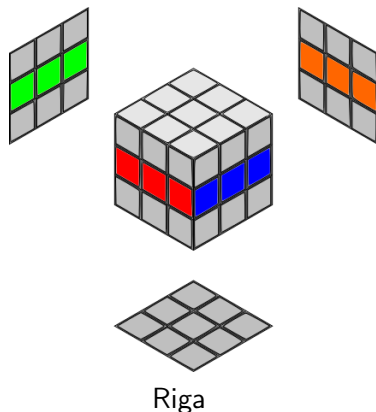
- ▶ Considereremo un cubo 3×3
- ▶ Ogni faccia (*side*) ha un colore standard a essa associato (vedi figura)
- ▶ Ognuno dei nove pezzi di ogni faccia è detto *facelet*
- ▶ Il cubo ha 3 colonne/righe (*columns/rows*), 3 colonne laterali (*lateral columns*), 4 angoli (*corners*) e 8 spigoli (*edges*)



Colonna

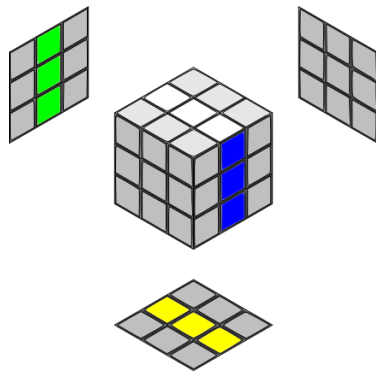
Alcune definizioni

- ▶ Considereremo un cubo 3×3
- ▶ Ogni faccia (*side*) ha un colore standard a essa associato (vedi figura)
- ▶ Ognuno dei nove pezzi di ogni faccia è detto *facelet*
- ▶ Il cubo ha 3 colonne/righe (*columns/rows*), 3 colonne laterali (*lateral columns*), 4 angoli (*corners*) e 8 spigoli (*edges*)



Alcune definizioni

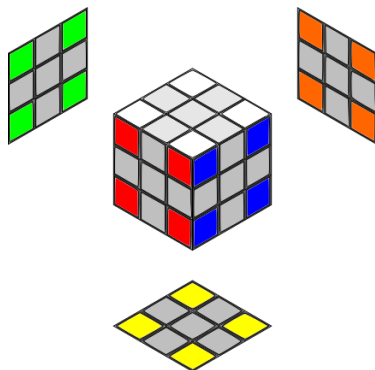
- ▶ Considereremo un cubo 3×3
- ▶ Ogni faccia (*side*) ha un colore standard a essa associato (vedi figura)
- ▶ Ognuno dei nove pezzi di ogni faccia è detto *facelet*
- ▶ Il cubo ha 3 colonne/righe (*columns/rows*), 3 colonne laterali (*lateral columns*), 4 angoli (*corners*) e 8 spigoli (*edges*)



Colonna laterale

Alcune definizioni

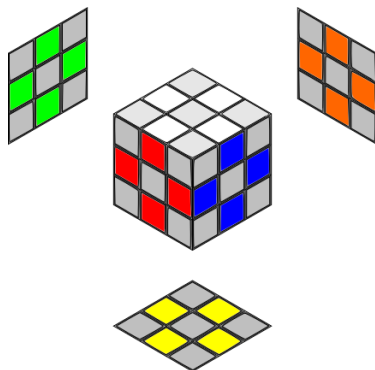
- ▶ Considereremo un cubo 3×3
- ▶ Ogni faccia (*side*) ha un colore standard a essa associato (vedi figura)
- ▶ Ognuno dei nove pezzi di ogni faccia è detto *facelet*
- ▶ Il cubo ha 3 colonne/righe (*columns/rows*), 3 colonne laterali (*lateral columns*), 4 angoli (*corners*) e 8 spigoli (*edges*)



Angolo

Alcune definizioni

- ▶ Considereremo un cubo 3×3
- ▶ Ogni faccia (*side*) ha un colore standard a essa associato (vedi figura)
- ▶ Ognuno dei nove pezzi di ogni faccia è detto *facelet*
- ▶ Il cubo ha 3 colonne/righe (*columns/rows*), 3 colonne laterali (*lateral columns*), 4 angoli (*corners*) e 8 spigoli (*edges*)



Spigolo

Il problema

Riarrangia il cubo (ruotando righe, colonne e/o colonne laterali) finché tutte le faccette su ogni faccia non hanno lo stesso colore.

Notazione di Singmaster

- ▶ Ogni faccia è descritta da una lettera: **F** (Front), **B** (Back), **U** (Up), **D** (Down), **L** (Left), **R** (Right)
- ▶ Ogni mossa può essere vista come una rotazione di un quarto di giro di una faccia in senso orario (N.B.: si assume che il solutore abbia la faccia di fronte a sé): **U** = ruota la faccia "Up" di un quarto di giro in senso orario
- ▶ Il simbolo ' indica una rotazione in senso antiorario
- ▶ Le rotazioni di righe/colonne/colonne laterali centrali sono denotate da **M** (*middle* — livello fra L e R), **E** (*equator* — livello fra U e D), **S** (*standing* — livello fra F e B)
- ▶ Per denotare le rotazioni del cubo si usano altre lettere: **X** (rotazione su R), **Y** (rotazione su U), **Z** (rotazione su F)

Notazione di Singmaster



F



F'



B



B'



U



U'



D



D'



L



L'



R



R'



M



M'



E



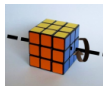
E'



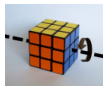
S



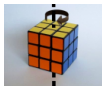
S'



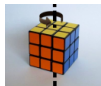
X



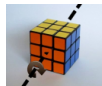
X'



Y



Y'



Z



Z'

Il nostro modello

- ▶ Il cubo è memorizzato in un oggetto `RubikCubeModel`
- ▶ Ogni faccia è memorizzata in un array 2×2 ; le righe/colonne sono numerate dall'alto verso il basso e da sinistra a destra (supponendo che il solutore abbia la faccia di fronte)
- ▶ `getSide` determina la faccia che in tale momento ha il colore dato
- ▶ `getFace` recupera il colore di una facelet
- ▶ Altri metodi autoesplicativi: `get3DEdge` (per gli spigoli), `get3DEdgeFacelet` (facelet di uno spigolo), `getCorner`, `getCornerFacelet`
- ▶ Metodi `rotate*` per ruotare il cubo
- ▶ Test standard: `isInStandardConfiguration`, `isWithSaneColors`, `isSolved`, `isCornerInPlace`, `isCornerInPlaceMaybeFlipped`, `isEdgeInPlace`, `isEdgeInPlaceMaybeFlipped`

Mosse di Singmaster

- ▶ Sono state implementate le mosse standard di Singmaster
- ▶ Ogni mossa (per motivi di astrazione) è una sottoclasse di Move
- ▶ Il costruttore accetta come parametri il modello del cubo (in modo che il cubo originale rimanga inalterato) e un parametro reversed (per sapere se la mossa è diretta o inversa)
- ▶ Per applicare una mossa, basta crearla e chiamare perform/reverse:

```
(new B(m, reversed)).perform();
```

- ▶ Ogni mossa genera un evento per comunicare i cambiamenti all'interfaccia

Strategie di risoluzione

- ▶ Sono sottoclassi di `ResolutionStrategy`
- ▶ Accettano un cubo (`RubikCubeModel`) e restituiscono una lista di mosse da eseguire per risolvere il problema (`getNextMoves`)

Pathfinding

- ▶ Possiamo rappresentare i possibili svolgimenti di una partita con un grafo i cui nodi sono la configurazione del cubo in un dato momento; due nodi sono collegati se e solo se ci si può recare da una configurazione a un'altra con una sola mossa
- ▶ L'idea alla base della maggior parte degli algoritmi di risoluzione del cubo è quella di trovare un cammino su tale albero avente origine nella radice (configurazione iniziale) e che termini nel cubo risolto

- ▶ Mantengo due liste: una (*open list*) che contiene i nodi ancora da valutare, un'altra (*closed list*) per i nodi già valutati
- ▶ Fisso una funzione *costo* per ogni nodo: esso deve essere, intuitivamente, tanto minore quanto minore è il “disordine” rispetto al cubo risolto
- ▶ Calcolo per ogni nodo N un indice

$$f(N) = g(N) + h(N)$$

dove $g(N)$ è il costo minimo dei nodi nella closed list e $h(N)$ è una stima del costo del nodo N (*euristica*)

- ▶ A ogni passo sposto il nodo considerato dalla open alla closed list (ad eccezione del caso in cui $g(N)$ diminuisca) e genero i suoi successori (tenendo traccia di tale legame)
- ▶ Al termine, estraendo il nodo con il minimo $f(N)$ e seguendo i genitori ho la sequenza di mosse cercata (al contrario)

$A^*(N, goalNode)$

```

1  openList = { N }
2  closedList =  $\emptyset$ 
3   $g(N) = 0$ 
4   $f(N) = h(N)$ 
5  while openList  $\neq \emptyset$ 
6      currentNode = Extract-Min- $f$ (openList)
7      if currentNode == goalNode
8          path =  $\langle currentNode \rangle$ 
9          while currentNode.parent  $\neq$  nil
10             currentNode = currentNode.parent
11             Append(path, currentNode)
12             return path
13     openList = openList  $\cap$  { currentNode }
14     closedList = closedList  $\cup$  { currentNode }
15     for ogni nodo  $N'$  vicino di currentNode
16         if currentNode  $\notin$  closedList
17             tentativeG =  $g(currentNode) + \text{Distance}(currentNode, N')$ 
18             if  $N' \notin openList$  o tentativeG <  $g(N')$ 
19                  $N'.parent = currentNode$ 
20                  $g(N') = tentativeG$ 
21                  $f(N') = g(N') + h(N')$ 
22                 if  $N' \notin openList$ 
23                     openList = openList  $\cup N'$ 
24     error "Nessuna soluzione trovata"

```


IDA*

- ▶ A* ha un difetto: richiede di esplorare tutto l'albero
- ▶ Non fattibile per il cubo di Rubik ($\sim 4,3 \times 10^{19}$ configurazioni possibili!)
- ▶ Basta non analizzare i rami per cui non crediamo di ottenere risultati
- ▶ IDA* fa questo: per ogni nodo, se $f(N)$ è maggiore di un certo valore limite che fissiamo, pota il ramo

IDA*

IDA*(N , $goalNode$)

1 $IDABound = h(N)$

2 **while** true

3 $t = \text{search}(N, 0, IDABound, goalNode)$

4 **if** $t == \text{found}$

5 **return** found

6 **if** $t == \infty$

7 **return** not-found

8 $IDABound = t$

IDA*

search($N, g, IDABound, goalNode$)

```
1   $f(N) = g + h(N)$ 
2  if  $f > IDABound$ 
3      return  $f(N)$ 
4  if  $N == goalNode$ 
5      return found
6   $min = \infty$ 
7  for ogni nodo  $N'$  vicino di  $N$ 
8       $t = \text{search}(N', g + \text{cost}(N, N'), IDABound)$ 
9      if  $t == \text{found}$ 
10         return found
11     if  $t < min$ 
12          $min = t$ 
13 return  $min$ 
```

Fissare un'euristica

- ▶ Rimane il problema di stabilire una buona $h(N)$
- ▶ Preferibilmente tale da soddisfare le seguenti proprietà:
 - ammissibile:** per ogni nodo, $h(N)$ è minore o uguale del costo effettivo per raggiungere N (per cui A^* non restituisce mai una soluzione subottimale)
 - consistente:** vale 0 in un nodo obiettivo e preserva una disuguaglianza triangolare: per ogni nodo N con figlio D , $h(N) \leq h(D) + \text{costo path da } N \text{ a } D$ (così in A^* , ogni volta in cui esamino un nodo, so già che il costo per raggiungerlo è il minimo possibile; non devo ricalcolarlo se trovo un path a costo minore)
- ▶ Nota: un'euristica consistente è automaticamente ammissibile

L'algoritmo di Thistlethwaite

- ▶ Thistlethwaite scoprì che era possibile dividere le configurazioni in cinque gruppi a seconda delle mosse da usare per risolvere il cubo:

$$G_0 = \langle L, R, F, B, U, D \rangle$$

$$G_1 = \langle L, R, F, B, U^2, D^2 \rangle$$

$$G_2 = \langle L, R, F^2, B^2, U^2, D^2 \rangle$$

$$G_3 = \langle L^2, R^2, F^2, B^2, U^2, D^2 \rangle$$

$$G_4 = \{1\}$$

- ▶ Si noti che ogni gruppo è chiaramente incluso nel precedente e che l'ultimo gruppo comprende il cubo risolto
- ▶ Idea: portare il cubo da una configurazione risolubile con tutte le mosse possibili (primo gruppo) in una risolubile solamente con mosse appartenenti al secondo gruppo, quindi al terzo. . .

L'algoritmo di Thistlethwaite

- ▶ Implementazione originaria: *lookup tables* di notevoli dimensioni calcolate a mano (!)
- ▶ Esaminando le configurazioni possibili si può ricavare un buon coefficiente euristico

L'algoritmo di Thistlethwaite e l'euristica

- ▶ Idea: partire dalla distanza di Manhattan tra lo spigolo/l'angolo esaminato e quelli nella posizione standard con i medesimi colori, quindi applicare opportuni fattori correttivi (ad es. incrementare il coefficiente se lo spigolo/angolo è molto vicino alla posizione corretta)
- ▶ Gli incrementi sono decisi sperimentalmente e/o aiutandosi con le tabelle viste in precedenza
- ▶ Si veda il metodo `getHeuristicCoefficient` in `IDAStar.java`

L'algoritmo a due fasi di Kociemba

- ▶ Osservazione: dato un cubo risolto, quelli ottenibili non usando le mosse R, R', L, L', F, F', B o B' mantengono invariati l'orientamento di spigoli e angoli (tale gruppo si denota con $G_{K1} = \langle U, D, R^2, L^2, F^2, B^2 \rangle$)
- ▶ Prima fase: si riconduce il cubo a uno di quelli appartenenti a G_{K1} con IDA* e un'euristica h_1 (nell'implementazione di Kociemba questa è implementata con una lookup table che consente un lookahead fino a 12 mosse). I cubi sono descritti da triple (orientamento angoli, orientamento spigoli, posizioni spigoli UD senza tener conto dell'ordine) che sono pari a (0,0,0) se e solo tali cubi appartengono a G_{K1} .

L'algoritmo a due fasi di Kociemba

- ▶ Seconda fase: usando solo mosse di G_{K1} si permutano spigoli e angoli per ottenere la disposizione corretta
- ▶ Si adottano triple (permutazione angoli, permutazione spigoli, coordinate spigoli UD); (0,0,0) se il cubo è risolto
- ▶ Depth-first search o IDA* (a seconda dei limiti di memoria/spazio su disco)

L'algoritmo a due fasi di Kociemba

Alcuni solutori e spiegazioni approfondite:

<http://kociemba.org/cube.htm>

L'algoritmo di Singmaster

- ▶ Risolve il cubo “per strati” in diverse fasi:
 1. si sposta la faccia bianca in alto
 2. si posizionano prima gli spigoli e poi gli angoli del livello superiore
 3. si posizionano gli spigoli del livello centrale
 4. si completa il livello inferiore
- ▶ Ampio uso di mosse del tipo $ABA'B'$ (sposto uno spigolo/angolo lasciando invariati gli altri)
- ▶ Molto lineare