Trabalho de Formatura Supervisionado

Soluções Eficientes para o Cubo Mágico

Aluno: Walter Pereira Rodrigues de Souza

Supervisor: Profa. Nina S. T. Hirata

Departamento de Ciência da Computação

Instituto de Matemática e Estatística

Universidade de São Paulo

São Paulo, 1 de dezembro de 2011

Sumário

1 Introdução **1**

2 Mecanismo **2**

3 Notação **4**

3.1 Faces 4

3.2 Peças 4

3.3 Adesivos 5

3.4 Movimentos 5

3.5 Permutação 6

3.6 Orientação 7

3.7 Combinação 9

4 Representação de estados **11**

4.1 Explícita 11

4.2 Por índices 13

5 Busca **16**

5.1 Tamanho do espaço de busca 16

5.2 Algoritmo de Korf 16

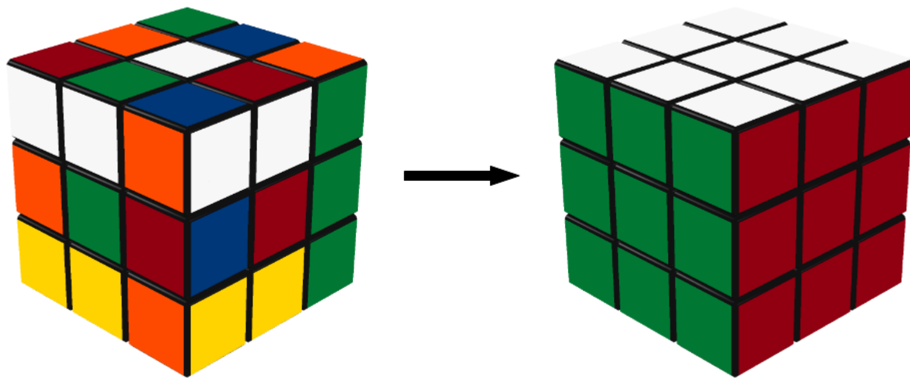
5.3 Algoritmo de Kociemba 19

6 Conclusão **21**

1 Introdução

O cubo mágico é um quebra-cabeça combinacional inventado em 1974 pelo arquiteto húngaro Ernö Rubik. Sua simplicidade e surpreendente dificuldade até hoje atrai e fascina pessoas de todas as idades.

O brinquedo é formado por 27 peças cúbicas dispostas na configuração $3 \times 3 \times 3$. No seu estado original, cada lado possui uma única cor (geralmente branco, amarelo, laranja, vermelho, verde e azul). Cada bloco $3 \times 3 \times 1$ pode ser movimentado independentemente do resto do cubo, embaralhando suas cores. O objetivo é trazer um estado aleatório ao estado original.



À primeira vista, o problema não parece tão difícil, mas basta tentar resolvê-lo sem nenhuma estratégia para se convencer do contrário. A explicação é muito simples: existem mais de 43 quinquilhões de posições diferentes e apenas uma é a correta.

Este trabalho descreve dois algoritmos que resolvem estados aleatórios do cubo mágico.

2 Mecanismo

A estrutura do cubo mágico é composta por três tipos diferentes de peças: centros, meios e quinas.

Os seis centros são ligados a um núcleo de forma que suas posições relativas nunca mudam, ou seja, o centro branco está sempre oposto ao amarelo, o laranja ao vermelho e o verde ao azul.

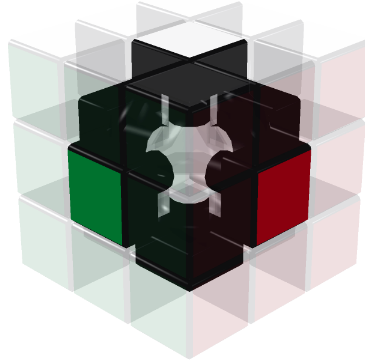


Figura 1: Centros

Entre cada par de centros adjacentes existe um meio. Os meios possuem duas faces à mostra, cada uma de uma cor diferente.

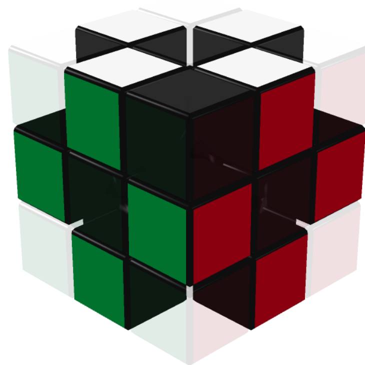


Figura 2: Meios

Entre cada trio de meios adjacentes existe uma quina. As quinas possuem três faces à mostra, cada uma de uma cor diferente.

Figura 3: Quinas

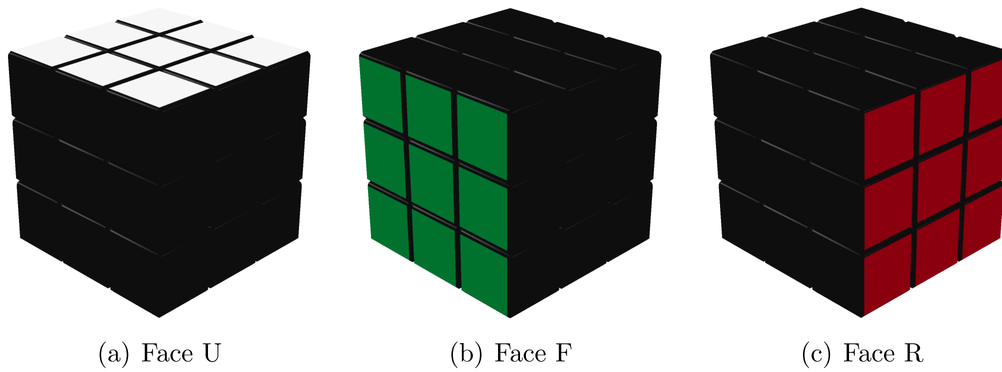
Cada bloco $3 \times 3 \times 1$ pode ser girado 90°, 180° ou 270°. Estes movimentos trocam apenas centro por centro, meio por meio e quina por quina, portanto, cada tipo de peça possui sua posição definida no cubo.

3 Notação

Antes de descrever os algoritmos, vamos definir alguns conceitos e nomenclatura.

3.1 Faces

As seis faces do cubo mágico são chamadas de U (cima), D (baixo), L (esquerda), R (direita), F (frente) e B (trás).



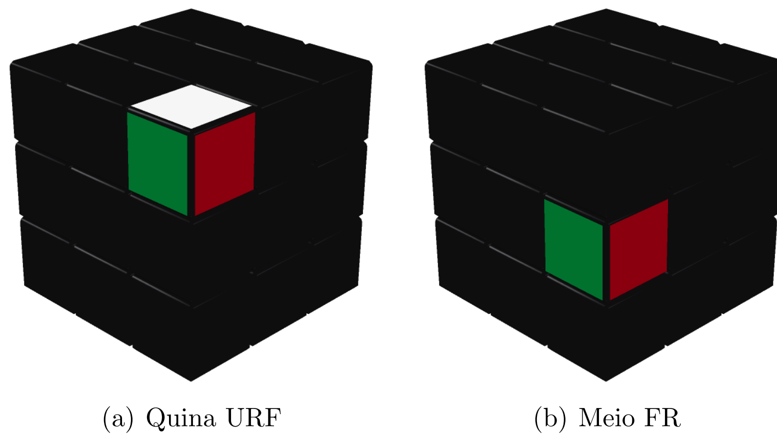
(a) Face U (b) Face F (c) Face R

Figura 4: Faces

A face D é oposta a U, R é oposta a L e B é oposta a F.

3.2 Peças

Os nomes das peças são formados pelos nomes das faces que as compõem. Por exemplo, o meio que pertence à face F e R é chamado de FR ou RF; a quina que pertence às faces U, R e F é chamada de URF, RFU ou FUR.

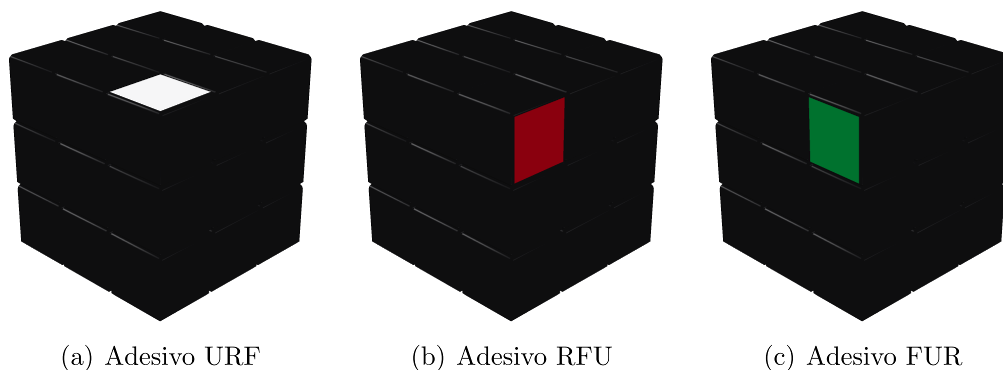


(a) Quina URF (b) Meio FR

Figura 5: Peças

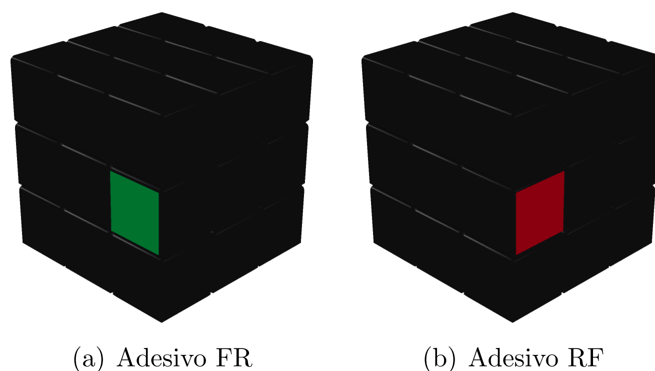
3.3 Adesivos

Os adesivos são nomeados de acordo com a peça e a face onde está. Por exemplo, a peça FR tem o adesivo FR na face F e o adesivo RF na face R; a peça URF tem o adesivo URF na face U, o adesivo RFU na face R e o adesivo FUR na face F.



(a) Adesivo URF (b) Adesivo RFU (c) Adesivo FUR

Figura 6: Adesivos de quinas

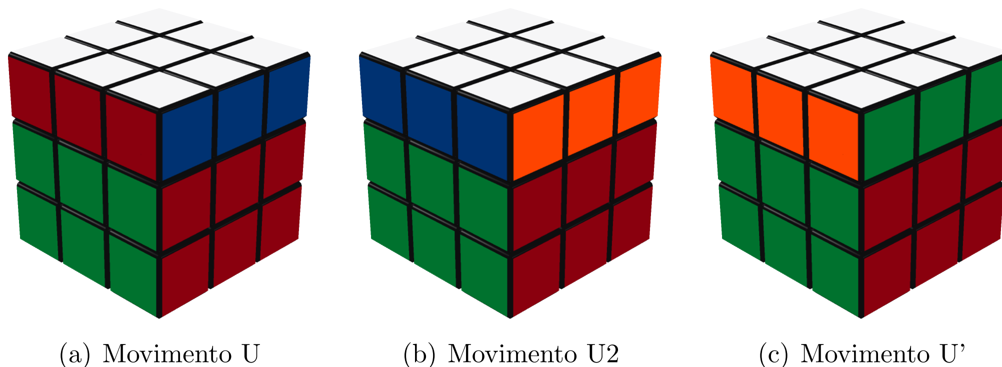


(a) Adesivo FR (b) Adesivo RF

Figura 7: Adesivos de meios

3.4 Movimentos

O movimento de 90° em sentido horário numa face recebe o mesmo nome da face. O movimento de 180° recebe o nome da face seguido de 2 (dois). O movimento de 90° em sentido anti-horário recebe o nome da face seguido de ' (linha).



(a) Movimento U (b) Movimento U2 (c) Movimento U'

Figura 8: Movimentos

3.5 Permutação

Permutação é o conceito relacionado a posição das peças de um estado específico do cubo mágico.

Como exemplo, vamos usar o estado que obtemos ao aplicar o movimento F' sobre o cubo resolvido.

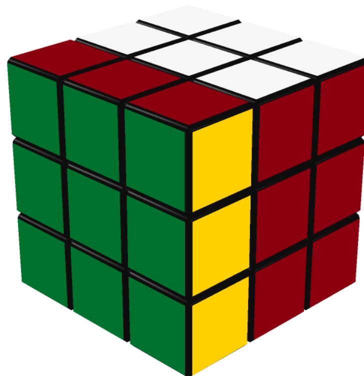


Figura 9: Movimento F'

Ignorando a orientação das peças, observamos a seguinte movimentação:

A posição da quina UFL recebe a quina URF, que recebe a DFR, que recebe a DLF, que finalmente recebe a UFL. A posição do meio UF recebe o meio FR, que recebe o DF, que recebe o FL, que recebe o UF, fechando o ciclo.

ULB	**UBR**	**URF**	**UFL**	**DBL**	**DRB**	**DFR**	**DLF**
ULB	UBR	DFR	URF	DBL	DRB	DLF	UFL

UB	**UR**	**UF**	**UL**	**BL**	**BR**	**FR**	**FL**	**DB**	**DR**	**DF**	**DL**
UB	UR	FR	UL	BL	BR	DF	UF	DB	DR	FL	DL

3.6 Orientação

É possível que uma peça esteja na sua posição correta, mas, mesmo assim, não esteja resolvida.

Observe as quinas UBR e URF no cubo abaixo:

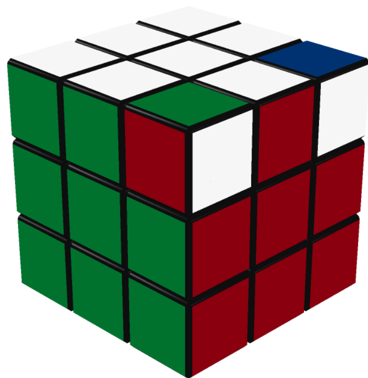


Figura 10: Quinas desorientadas

Poderíamos chegar a este estado pegando um cubo resolvido e girando a quina UBR em sentido anti-horário e a URF em sentido horário. Dizemos, então, que a quina UBR está desorientada em sentido anti-horário (-1), a quina URF em sentido horário (1) e todas as outras estão orientadas (0).

ULB	**UBR**	**URF**	**UFL**	**DBL**	**DRB**	**DFR**	**DLF**
0	-1	1	0	0	0	0	0

Determinar a orientação de uma quina é fácil quando ela está no seu local correto, mas precisamos saber a orientação de qualquer quina independentemente da sua posição.

Para isso vamos vamos escolher um adesivo por quina para servir de referência de orientação. São eles os brancos (ULB, UBR, URF, UFL) e os amarelos (DBL, DRB, DFR, DLF).

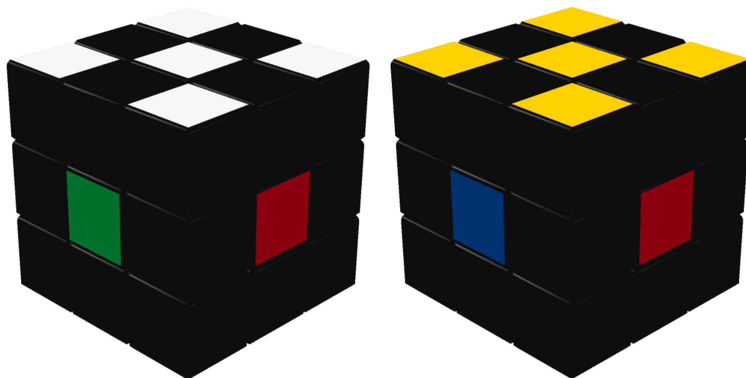
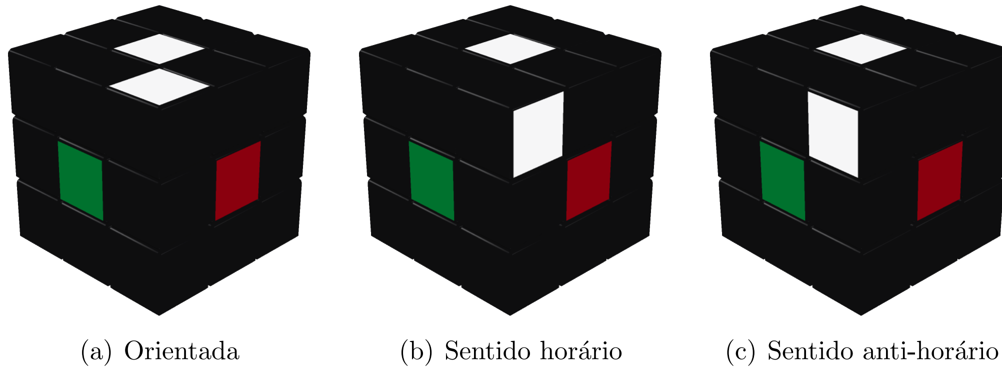


Figura 11: Adesivos de referência das quinas

Se o adesivo de referência estiver nas faces U ou D, a quina está orientada; se um giro anti-horário traz o adesivo de referência para a face U ou D, a quina está desorientada em sentido horário; caso contrário, a quina está desorientada em sentido anti-horário.



(a) Orientada (b) Sentido horário (c) Sentido anti-horário

Figura 12: Orientações de quinas

A orientação dos meios funciona de forma parecida, mas agora são apenas dois valores diferentes. Observe os meios UF e FR no cubo abaixo:

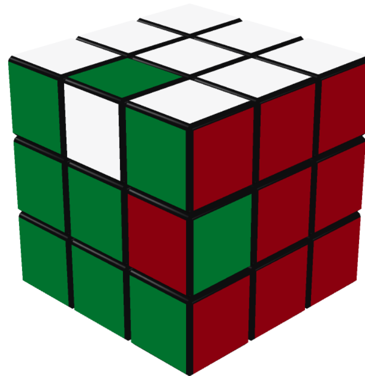


Figura 13: Meios desorientados

Ambos estão desorientados (1) e todos os outros estão orientados (0).

UB	**UR**	**UF**	**UL**	**BL**	**BR**	**FR**	**FL**	**DB**	**DR**	**DF**	**DL**
0	0	1	0	0	0	1	0	0	0	0	0

Os adesivos de referência dos meios são os brancos (UB, UR, UF, UL), os amarelos (DB, DR, DF, DL) e, para os meios que não possuem adesivos brancos ou amarelos, os verdes (FL, FR) e os azuis (BL, BR).

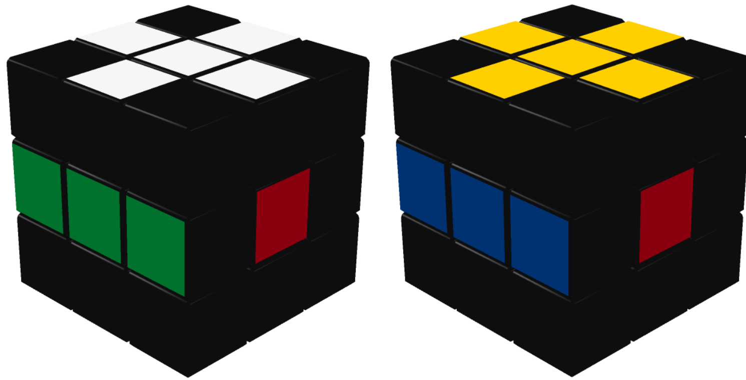
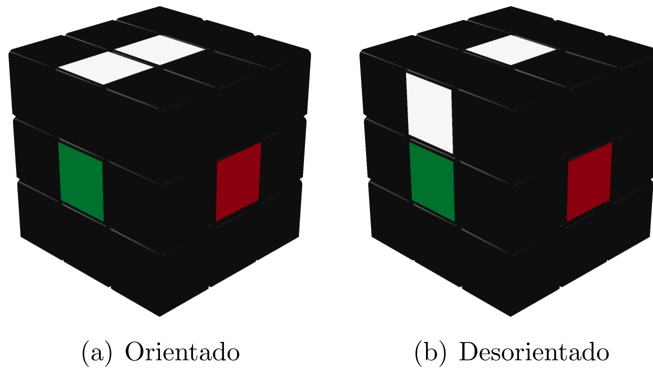


Figura 14: Adesivos de referência dos meios

Se o adesivo de referência estiver nas faces U, D, F ou B, o meio está orientado; caso contrário, está desorientado.



(a) Orientado (b) Desorientado

Figura 15: Orientações de meios

3.7 Combinação

O conceito de combinação está relacionado com a posição de um grupo de peças, sem se importar com a localização individual delas.

Digamos que estamos interessados nas quinas ULB, UBR, URF e UFL. A combinação dessas peças em um cubo resolvido é:

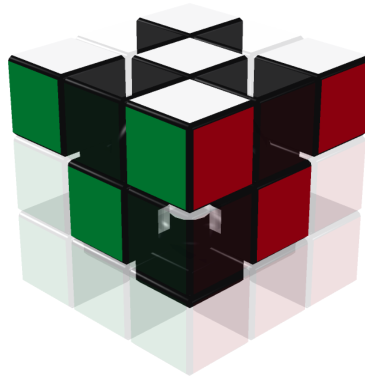


Figura 16: Meios desorientados

ULB	UBR	URF	UFL	DBL	DRB	DFR	DLF
1	1	1	1	0	0	0	0

As posições ULB, UBR, URF e UFL contêm as peças nas quais estamos interessados (1); todas as outras, não (0).

Se aplicarmos o movimento F, a quina URF vai para a posição DFR e a quina UFL vai para a posição URF. A nova combinação é a seguinte:

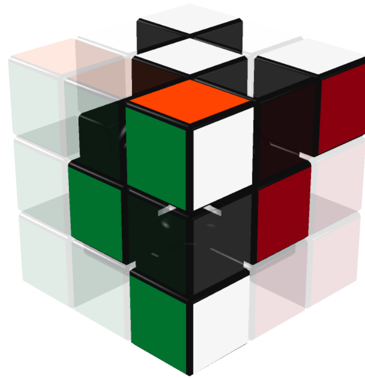


Figura 17: Meios desorientados

ULB	UBR	URF	UFL	DBL	DRB	DFR	DLF
1	1	1	0	0	0	1	0

As posições ULB, UBR, URF e DFR contêm as peças nas quais estamos interessados (1); todas as outras, não (0).

As combinações de meios funcionam exatamente da mesma forma.

4 Representação de estados

Agora que temos todos os conceitos necessários definidos, podemos iniciar o desenvolvimento dos solucionadores.

O primeiro passo é encontrar uma representação que torne possível a manipulação de estados do cubo mágico pelo computador.

4.1 Explícita

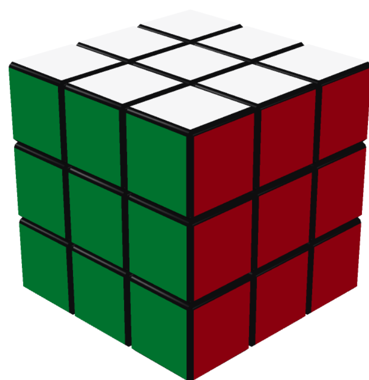
A forma mais óbvia de se representar um cubo mágico é armazenar diretamente a permutação/orientação das quinas e dos meios. Como os centros não precisam ser movimentados, não precisamos representá-los.

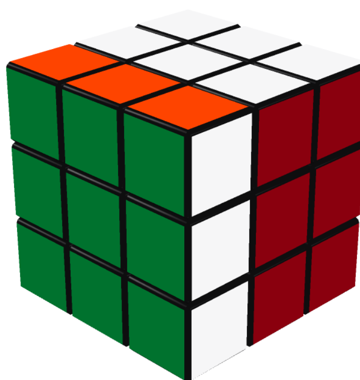
Para descrever as permutações, associaremos um número a cada peça (1 a 8 para quinas, 1 a 12 para meios). Também associaremos um número a cada orientação (0, 1 e 2 para quinas, 0 e 1 para meios).

ULB	**UBR**	**URF**	**UFL**	**DBL**	**DRB**	**DFR**	**DLF**
1	2	3	4	5	6	7	8

UB	**UR**	**UF**	**UL**	**BL**	**BR**	**FR**	**FL**	**DB**	**DR**	**DF**	**DL**
1	2	3	4	5	6	7	8	9	10	11	12

Podemos agora representar qualquer estado por uma tupla de quatro vetores de inteiros:


(a) Cubo resolvido


(b) Movimento F

```
1  -- cubo resolvido
2  id = {
3    pQuinas = { 1, 2, 3, 4, 5, 6, 7, 8 },
4    oQuinas = { 0, 0, 0, 0, 0, 0, 0, 0 },
5    pMeios = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 },
```

```
 6    oMeios = { 0, 0, 0, 0, 0, 0, 0, 0, 0,  0,  0,  0 },
 7  }
 8
 9  -- movimento F
10  f = {
11    pQuinas = { 1, 2, 4, 8, 5, 6, 3, 7 },
12    oQuinas = { 0, 0, 1, 2, 0, 0, 2, 1 },
13    pMeios = { 1, 2, 7, 11, 5, 6, 4, 8, 9, 10, 3, 12 },
14    oMeios = { 0, 0, 1,  1, 0, 0, 1, 0, 0,  0, 1,  0 },
15  }
```

Observem que representamos os estados como transformações sobre o cubo resolvido. Se usássemos qualquer outro, os resultados seriam os mesmos. Interpretar posições específicas como transformações nos permite *compor* estados:

```
 1  function mul(e1, e2)
 2    local pQuinas = {}
 3    local oQuinas = {}
 4    for i = 1, 8 do
 5      pQuinas[i] = e1.pQuinas[e2.pQuinas[i]]
 6      oQuinas[i] = (e1.oQuinas[e2.pQuinas[i]] + e2.oQuinas[i]) % 3
 7    end
 8
 9    local pMeios = {}
10    local oMeios = {}
11    for i = 1, 12 do
12      pMeios[i] = e1.pMeios[e2.pMeios[i]]
13      oMeios[i] = (e1.oMeios[e2.pMeios[i]] + e2.oMeios[i]) % 2
14    end
15
16    return {
17      pQuinas = pQuinas,
18      oQuinas = oQuinas,
19      pMeios = pMeios,
20      oMeios = oMeios,
21    }
22  end
```

A função `mul` compõe os estados `e1` e `e2`. O seu funcionamento é bem simples: as linhas 5 e 12 fazem a composição usual de permutações; as linhas 6 e 13 fazem a soma de orientações (módulo 3 para quinas e módulo 2 para meios).

4.2 Por índices

A representação explícita é bastante conveniente, porém, exige muita memória e a composição é computacionalmente cara.

Uma alternativa é usar as bijeções existentes entre as permutações/orientações/combinações e os inteiros para representar estados como tuplas de índices [Scherphuis, 2011].

```
1  -- permutação <-> [0, n! - 1]
2  function permutacao_para_indice(p) ... end
3  function indice_para_permutacao(i, n) ... end
4
5  -- orientação <-> [0, b^n - 1]
6  function orientacao_para_indice(o, b) ... end
7  function indice_para_orientacao(i, b, n) ... end
8
9  -- combinação <-> [0, C(n, k) - 1]
10 function combinacao_para_indice(c, k) ... end
11 function indice_para_combinacao(i, k, n) ... end
12
13 -- estado -> indices
14 function estado_para_indices(e)
15   return {
16     pQuinas = permutacao_para_indice(e.pQuinas),
17     oQuinas = orientacao_para_indice(e.oQuinas, 3),
18     pMeios = permutacao_para_indice(e.pMeios),
19     oMeios = orientacao_para_indice(e.oMeios, 2),
20   }
21 end
22
23 -- indices -> estado
24 function indices_para_estado(i)
25   return {
26     pQuinas = indice_para_permutacao(i.pQuinas, 8),
27     oQuinas = indice_para_orientacao(i.oQuinas, 3, 8),
28     pMeios = indice_para_permutacao(i.pMeios, 12),
29     oMeios = indice_para_orientacao(i.oMeios, 2, 12),
30   }
31 end
```

As funções `permutacao_para_indice`, `orientacao_para_indices` e `combinacao_para_indice` recebem um vetor de inteiros e devolvem o índice correspondente. As funções `indice_para_permutacao`, `indice_para_orientacao` e `indice_para_combinacao` fazem a operação inversa.

Da mesma forma, a função `estado_para_indice` recebe uma representação explícita de estado e devolve uma tupla de índices. A função `indice_para_estado` faz a operação inversa.

A representação por índices economiza memória (cada cubo ocupa apenas 4 inteiros), mas a composição dos estados não pode ser feita diretamente. Para contornar este problema iremos introduzir as tabelas de transição.

Tabela de transição são mapeamentos da forma (índice, movimento) → índice. Se construirmos uma tabela para permutação de quinas, uma para orientação de quinas, uma para permutação de meios e uma para orientação de meios, podemos manipular os estados da seguinte forma:

```
1  function mul(e, movimento)
2    return {
3      pQuinas = pQuinasTrans[e.pQuinas][movimento],
4      oQuinas = oQuinasTrans[e.oQuinas][movimento],
5      pMeios = pMeiosTrans[e.pMeios][movimento],
6      oMeios = oMeiosTrans[e.oMeios][movimento],
7    }
8  end
```

Reparem que o segundo parâmetro da função não recebe um estado qualquer, mas sim aqueles definidos na construção das tabelas.

A construção das tabelas de transição é bem simples. Como exemplo, veremos a tabela de orientação de quinas:

```
1  movimentos = {
2    ["U"]  = { pQuinas = { ... }, oQuinas = { ... }, ... },
3    ["U2"] = { pQuinas = { ... }, oQuinas = { ... }, ... },
4    ["U'"] = { pQuinas = { ... }, oQuinas = { ... }, ... },
5    ...                                                  },
6    ["B2"] = { pQuinas = { ... }, oQuinas = { ... }, ... },
7    ["B'"] = { pQuinas = { ... }, oQuinas = { ... }, ... },
8  }
9
10 oQuinasTrans = {}
11 for i = 0, 3^8 - 1 do
12   local estado = indices_para_estado({ oQuinas = i })
13
14   oQuinasTrans[i] = {}
15   for nome, mov in pairs(movimentos) do
16     oQuinasTrans[i][nome] =
17       estado_para_indices(mul(estado, mov)).oQuinas
18   end
19 end
```

O primeiro passo é definir quais estados serão suportados como segundo parâmetro da função `mul` (linhas 1-8). Geralmente usamos todos os movimentos de face (U, U2,

U', D, D2, D', L, L2, L', R, R2, R', F, F2, F', B, B2 e B'), mas também podemos usar subconjuntos diferentes dependendo do solucionador.

Para cada índice válido de orientação de quinas (linha 11), criamos um estado correspondente (linha 12). Agora, aplicamos os movimentos pré-determinados a este estado e armazenamos os índices resultantes na tabela (linhas 15-18).

A construção das outras tabelas é similar.

5 Busca

5.1 Tamanho do espaço de busca

Antes de entrar nos detalhes do algoritmo de busca, vamos calcular o número de estados válidos do cubo mágico.

Os fatores deste número são a permutação das quinas (8!), orientação das quinas (3^8), permutação dos meios (12!) e orientação dos meios (2^{12}):

$$519\ 024\ 039\ 293\ 878\ 272\ 000$$

Algumas restrições não permitem que alcancemos todos estes estados a partir do cubo resolvido.

São duas restrições de orientação: a orientação de sete quinas define a orientação da última (1/3); a orientação de onze meios define a orientação do último (1/2).

Uma restrição de permutação: a permutação das quinas deve casar com a permutação dos meios, ou seja, ambas devem ser pares ou ambas ímpares (1/2).

Dessa forma chegamos ao total de:

$$43\ 252\ 003\ 274\ 489\ 856\ 000$$

Para se ter uma ideia do tamanho deste número, imagine que uma pessoa resolva um estado aleatório do cubo mágico em 10 segundos em média. Se esta pessoa decidisse resolver todos os estados válidos, levaria no mínimo 1.37×10^{13} anos, aproximadamente mil vezes a idade estimada do universo.

5.2 Algoritmo de Korf

Em 1997, Richard Korf desenvolveu o primeiro algoritmo capaz de resolver otimamente instâncias aleatórias do cubo mágico [Korf, 1997].

Pelo tamanho do espaço de busca, é óbvio que precisamos guiar a busca de alguma forma. Para isso usaremos heurísticas, ou seja, funções que estimam a distância entre um estado qualquer e o estado objetivo.

As heurísticas do algoritmo de Korf são baseadas na solução ótima de subproblemas do cubo mágico, mais especificamente, na solução das quinas e na solução de conjuntos de seis meios.

Se considerarmos apenas as quinas, temos $8! \times 3^7 = 88179840$ estados diferentes. Este espaço de busca pode ser explorado em poucos segundos em um PC moderno.

```
1  iId = estado_para_indices(id)
```

```
 2
 3  quinasDist = {}
 4  quinasDist[iId.pQuinas][iId.oQuinas] = 0
 5
 6  local distancia = 0
 7  local visitados = 0
 8  while visitados < 8! * 3^7 do
 9    for i = 0, 8! - 1 do
10      for j = 0, 3^7 - 1 do
11        if quinasDist[i][j] == distancia then
12          for nome, _ in pairs(movimentos) do
13            local pProx = pQuinasTrans[i][nome]
14            local oProx = oQuinasTrans[j][nome]
15            if not quinasDist[pProx][oProx] then
16              quinasDist[pProx][oProx] = distancia + 1
17              visitados = visitados + 1
18            end
19          end
20        end
21      end
22    end
23  end
```

O código acima cria uma tabela com a distância entre cada estado válido das quinas e o estado resolvido usando uma busca em largura.

No primeiro passo, o estado resolvido é marcado com distância 0. No segundo passo, os vizinhos do estado resolvido são marcados com distância 1. No terceiro passo, os vizinhos dos estados de distância 1 são marcados com distância 2. Assim a busca prossegue até que todos os estados sejam alcançados.

A distribuição das distâncias é a seguinte:

0	1
1	18
2	243
3	2 874
4	28 000
5	205 416
6	1 168 516
7	5 462 528
8	20 776 176
9	45 391 616
10	15 139 616
11	64 736

Poderíamos fazer o mesmo com os meios, mas o espaço de busca tem tamanho $12! \times 2^{11} = 980995276800$, ou seja, muito grande para os computadores atuais. O que podemos fazer é considerar dois subconjuntos de seis meios, assim temos $2 \times \binom{12}{6} \times 6! \times 2^6 = 2 \times 42577920$ estados diferentes.

A distribuição das distâncias dos subconjuntos é a seguinte:

0	1
1	15
2	2 360
3	27 139
4	281 416
5	2 380 459
6	13 065 209
7	23 961 831
8	2 859 244
9	56

O algoritmo de busca utilizado é o IDA* (*Iterative deepening A**, A* com aprofundamento iterativo):

```
 1  function solucao(e)
 2    for i = 0, inf do
 3      local sol = {}
 4      if busca(e, i, sol) then
 5        return sol
 6      end
 7    end
 8  end
 9
10  function busca(e, prof, sol)
11    if prof == 0 then
12      return i == iId
13    end
14
15    if quinasDist[e.cQuinas][e.pQuinas][e.oQuinas] > prof or
16       meios1Dist[e.cMeios1][e.pMeios1][e.oMeios1] > prof or
17       meios2Dist[e.cMeios2][e.pMeios2][e.oMeios2] > prof then
18      return false
19    end
20
21    for nome, _ in pairs(movimentos) do
22      table.insert(sol, nome)
23      if busca(mul(e, nome), prof - 1, sol) then
```

```
24          return true
25        end
26        table.remove(sol)
27      end
28    end
```

O algoritmo IDA* realiza seguidas buscas em profundidade, sempre aumentando a profundidade máxima explorada até encontrar um caminho entre o nó original e o nó solução (linhas 1-8).

A função `busca` se parece com uma busca em profundidade comum, com exceção das linhas 15-19, que são responsáveis por podar a árvore de busca. Por exemplo, se a busca está em um nó que precisa de no mínimo 8 movimentos para ser resolvido, mas há apenas mais 6 níveis para explorar, este ramo pode ser descartado.

Como as heurísticas usadas são admissíveis, ou seja, nunca superestimam a distância real entre um nó e o nó resolvido, o comprimento do caminho encontrado pelo algoritmo de busca é mínimo.

5.3 Algoritmo de Kociemba

O algoritmo de Korf, apesar de eficiente, pode levar até mesmo horas para encontrar a solução de uma posição aleatória do cubo mágico em um computador atual. O algoritmo de Kociemba [Kociemba, 2011] oferece uma alternativa que encontra soluções quase ótimas, mas levando muito menos tempo.

A ideia principal do algoritmo de Kociemba é definir um objetivo intermediário e resolver o problema em dois passos, ou seja, pegar um cubo completamente embaralhado, encontrar uma sequência de movimentos que o coloque num subconjunto especial do espaço de busca e, finalmente, resolvê-lo.

O subconjunto especial em questão é aquele que contém apenas os estados que podem ser resolvidos utilizando somente os movimentos U, U2, U', D, D2, D', L2, R2, F2 e B2. O que caracteriza este subconjunto é que todas a peças estão orientadas e os meios BL, BR, FR e FL estão na sua combinação correta.

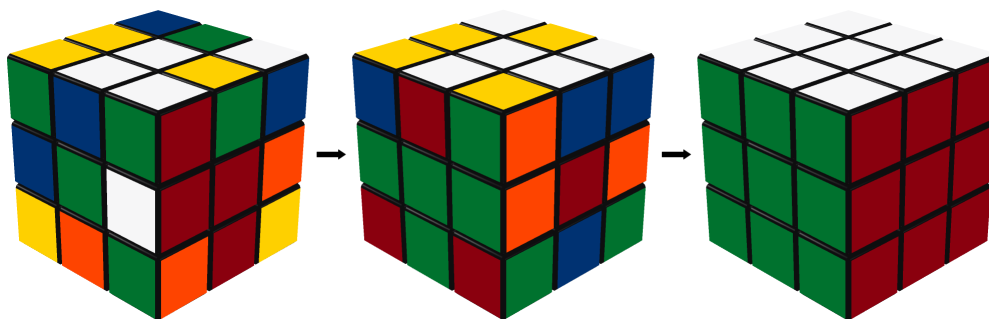


Figura 18: Solução em duas fases

A busca dentro de cada fase é idêntica à busca do algoritmo de Korf (ainda são usadas tabelas de transição, tabelas de distância e IDA*), mas um truque garante que a solução encontrada seja quase ótima.

A procura não acaba quando é encontrada uma solução ótima da primeira fase e uma solução ótima da segunda fase. Por exemplo, digamos que um estado tenha solução de primeira fase de comprimento 12 e solução de segunda fase de comprimento 11, totalizando 23. Se encontrarmos uma solução da primeira fase com comprimento 13, mas que leva a uma solução da segunda fase com comprimento 7, teremos uma solução de 20 movimentos.

```
 1  function solucao(e)
 2    local comprimento = inf
 3
 4    for sol1 in solucoes_fase1(e) do
 5      local e2 = e
 6      for _, nome in ipairs(sol1) do
 7        e2 = mul(e2, movimentos[nome])
 8      end
 9
10      local max = math.min(comprimento - table.maxn(sol1), 12)
11      for sol2 in solucoes_fase2(e2, max) do
12        comprimento = table.maxn(sol1) + table.maxn(sol2)
13
14        local sol = {}
15        for _, nome in ipairs(sol1) do
16          table.insert(sol, nome)
17        end
18        for _, nome in ipairs(sol2) do
19          table.insert(sol, nome)
20        end
21
22        yield sol
23
24        break
25      end
26    end
27  end
```

6 Conclusão

Este trabalho apresentou dois algoritmos eficientes para solução do cubo mágico.

As ideias discutidas podem ser aplicadas com pequenas modificações para resolver outros tipos de quebra-cabeças, como o $2 \times 2 \times 2$, o *skewb*, o *pyraminx*, o *square-1*, etc..

Referências

[Kociemba, 2011] Kociemba, H. (2011). Solve rubik's cube with cube explorer. `http://kociemba.org/cube.htm`.

[Korf, 1997] Korf, R. E. (1997). Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705.

[Scherphuis, 2011] Scherphuis, J. (2011). Jaap's puzzle page. `http://www.jaapsch.net/puzzles`.