

Use an ILP to Solve the Cube of Rubik

March 21, 2022

1 Optimal solution of Rubik's cube with Mixed-Integer Linear Programming

Combinatorial Optimization course, FEE CTU in Prague. Created by [Industrial Informatics Department](#).

In this assignment, we will demonstrate the use of ILP formalism to find the shortest sequence of moves that will solve Rubik's cube from any feasible initial configuration. The inspiration for the model is taken from <http://www.m-hikari.com/imf-password2009/45-48-2009/aksopIMF45-48-2009-2.pdf> where some mistakes were corrected and the model was a bit improved.

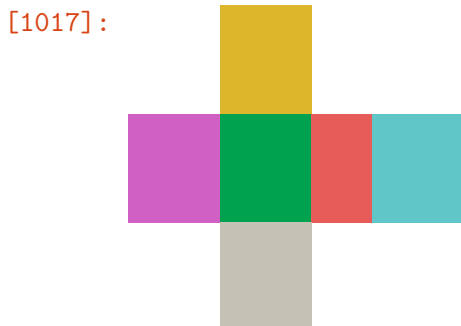
For visualisation and performing moves to the cube, we use nifty package `pycuber` which can be installed simply by

```
pip install pycuber
```

```
[1016]: import pycuber as pc
import gurobipy as g
import numpy as np
import itertools as it
```

This is how a solved cube looks like.

```
[1017]: c = pc.Cube()
c
```



Now, in each turn we can apply 18 different moves. We will randomly scramble our cube by some of the moves:

```
[1018]: moves_mapping = ["B'", "B", "S", "S'", "F", "F'", "U'", "U", "E", "E'", "D", "D'",
    ↪ "D'", "L'", "L", "M'", "M", "R", "R'"]

c = pc.Cube()

# random scramble
alg = ' '.join(np.random.choice(moves_mapping, 5, replace=True))
fc = c.copy()
fc(alg)
print(alg)
c(alg)
```

B' L F R F'



Now, we will prepare some data for our ILP model. We need to somehow index the individual subcubes to represent the initial state of the cube.

```
[1019]: cubes_idx = np.zeros(shape=(6, 3, 3), dtype=int)

counter = 1
for i in range(3):
    for j in range(3):
        cubes_idx[0, i, j] = counter
        cubes_idx[5, i, j] = counter + 45
        counter += 1
for j in range(3):
    for k in range(4):
        cubes_idx[1+k, 0, j] = 10 + j + k*3
        cubes_idx[1+k, 1, j] = 10 + j + k*3 + 12
        cubes_idx[1+k, 2, j] = 10 + j + k*3 + 24

faces = []
for f in range(6):
    faces.append([])
    for i in range(3):
        for j in range(3):
            faces[f] += [cubes_idx[f, i, j]]
```

```

faces_letters = 'U L F R B D'.split()
colors = ['red', 'yellow', 'green', 'white', 'orange', 'blue']
init_pattern = []
for face_idx, face in enumerate(faces_letters):
    a = c.get_face(face)
    for i in range(3):
        for j in range(3):
            init_pattern += [(cubes_idx[face_idx, i, j], colors.index(a[i][j]).
↪ colour) + 1)]

```

1.1 ILP model

Finally, we build and solve the model.

```

[1026]: G = [
    (1,1,18), (1,2,30), (1,3,42), (1,10,3), (1,22,2),
    (1,34,1), (1,18,54), (1,30,53), (1,42,52), (1,52,10), (1,53,22),
    (1,54,34), (1,19,43), (1,20,31), (1,21,19), (1,31,44), (1,33,20),
    (1,43,45), (1,44,33), (1,45,21), (3,4,17), (3,5,29), (3,6,41), (3,11,6), ↪
↪ (3,23,5), (3,35,4), (3,17,51), (3,29,50), (3,41,49),
    (3,49,11), (3,50,23), (3,51,35), (5,7,16), (5,8,28), (5,9,40),
    (5,12,9), (5,24,8), (5,36,7), (5,16,48), (5,28,47), (5,40,46),
    (5,46,12), (5,47,24), (5,48,36), (5,13,15), (5,14,27), (5,15,39),
    (5,25,14), (5,27,38), (5,37,13), (5,38,25), (5,39,37), (7,10,13),
    (7,11,14), (7,12,15), (7,13,16), (7,14,17), (7,15,18), (7,16,19),
    (7,17,20), (7,18,21), (7,19,10), (7,20,11), (7,21,12), (7,3,1),
    (7,6,2), (7,9,3), (7,2,4), (7,8,6), (7,1,7), (7,4,8), (7,7,9),
    (9,22,25), (9,23,26), (9,24,27), (9,25,28), (9,26,29), (9,27,30),
    (9,28,31), (9,29,32), (9,30,33), (9,31,22), (9,32,23), (9,33,24),
    (11,34,37), (11,35,38), (11,36,39), (11,37,40), (11,38,41),
    (11,39,42), (11,40,43), (11,41,44), (11,42,45), (11,43,34),
    (11,44,35), (11,45,36), (11,46,48), (11,47,51), (11,48,54),
    (11,49,47), (11,51,53), (11,52,46), (11,53,49), (11,54,52),
    (13,1,45), (13,4,33), (13,7,21), (13,13,1), (13,25,4), (13,37,7),
    (13,21,52), (13,33,49), (13,45,46), (13,46,13), (13,49,25),
    (13,52,37), (13,10,34), (13,11,22), (13,12,10), (13,22,35),
    (13,24,11), (13,34,36), (13,35,24), (13,36,12), (15,2,44), (15,5,32),
    (15,8,20), (15,14,2), (15,26,5), (15,38,8), (15,20,53), (15,32,50),
    (15,44,47), (15,47,14), (15,50,26), (15,53,38), (17,3,43), (17,6,31),
    (17,9,19), (17,15,3), (17,27,6), (17,39,9), (17,19,54), (17,31,51),
    (17,43,48), (17,48,15), (17,51,27), (17,54,39), (17,16,18),
    (17,17,30), (17,18,42), (17,28,17), (17,30,41), (17,40,16),
    (17,41,28), (17,42,40)
]

```

```

# in theory, with at most 26 moves you can solve the cube from any initial
↳ configuration: http://cube20.org/qtm/
max_moves = 6

max_moves += 1
m = g.Model()
y = m.addVars(18, max_moves, vtype=g.GRB.BINARY, name='y')
x = m.addVars(54, max_moves, vtype=g.GRB.INTEGER, lb=1, ub=6, name='x')
moves_used = m.addVar(vtype=g.GRB.CONTINUOUS, ub=max_moves, obj=1)

# objective: minimize the number of used moves
m.addConstrs(moves_used >= g.quicksum((t+1)*y[i, t] for i in range(18)) for t
↳ in range(max_moves))

for t in range(max_moves-1):
    for k, i, j in G:
        # if we perform move k at time t, then colors of the affected cubes
↳ must change accordingly
        # beware! this is not an ILP constraint. but can be translated to a one
↳ with a big-M
        m.addConstr((y[k-1, t] == 1) >> (x[i-1, t] == x[j-1, t+1]))

        # inverse to the k move
        m.addConstr((y[k+1-1, t] == 1) >> (x[j-1, t] == x[i-1, t+1]))

        # non-affected cubes must remain the same
        m.addConstr(x[i-1, t] - 6*(y[k-1, t] + y[k+1-1, t] +
↳ g.quicksum(y[l-1, t] + y[l+1-1, t] for l, m,
↳ n in G if m==i and k!=1)) <= x[i-1, t+1]
        )
        m.addConstr(x[i-1, t+1] <= x[i-1, t] + 6*(y[k-1, t] + y[k+1-1, t] +
↳ g.quicksum(y[l-1, t] + y[l+1-1, t] for l, m,
↳ n in G if m==i and k!=1))
        )

# final state conditions - all cubes in every face must have the same colors
for f in range(6):
    for i in faces[f]:
        for j in faces[f]:
            if i > j:
                m.addConstr(x[i-1, max_moves-1] == x[j-1, max_moves-1])

# set initial cube configuration
for cidx, color in init_pattern:

```

```

m.addConstr(x[cidx-1, 0] == color)

# one move at the time
m.addConstrs(g.quicksum(y[i, t] for i in range(18)) <= 1 for t in
↳range(max_moves))

# redundant constraints:
for t in range(max_moves-1):
    # forbid mirror moves (gives solver extra information) +
    # symmetry breaking: do not use sequence of two moves of the second kind (i.
↳e., prime moves)
    # if it has to do it, it will achieve the same effect with two moves of the
↳first kind
    m.addConstrs(y[2*k+1, t] + y[2*k, t+1] <= 1 for k in range(9))
    m.addConstrs(y[2*k, t] + y[2*k+1, t+1] + y[2*k+1, t] <= 1 for k in range(9))

# solve the problem
m.params.mipfocus = 1 # focus on feasibility
m.optimize()

```

```

[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11, 12, 22, 23, 24, 34, 35, 36], [13, 14, 15,
25, 26, 27, 37, 38, 39], [16, 17, 18, 28, 29, 30, 40, 41, 42], [19, 20, 21, 31,
32, 33, 43, 44, 45], [46, 47, 48, 49, 50, 51, 52, 53, 54]]

```

Changed value of parameter mipfocus to 1

Prev: 0 Min: 0 Max: 3 Default: 0

Gurobi Optimizer version 9.0.0 build v9.0.0rc2 (mac64)

Optimize a model with 2264 rows, 505 columns and 15703 nonzeros

Model fingerprint: 0x8c139283

Model has 1872 general constraints

Variable types: 1 continuous, 504 integer (126 binary)

Coefficient statistics:

Matrix range [1e+00, 7e+00]

Objective range [1e+00, 1e+00]

Bounds range [1e+00, 7e+00]

RHS range [1e+00, 6e+00]

Presolve added 2083 rows and 0 columns

Presolve removed 0 rows and 72 columns

Presolve time: 0.10s

Presolved: 4347 rows, 433 columns, 15294 nonzeros

Variable types: 0 continuous, 433 integer (108 binary)

Root relaxation: objective 6.617673e-01, 2585 iterations, 0.07 seconds

Nodes		Current Node			Objective Bounds			Work	
Expl	Unexpl	Obj	Depth	IntInf	Incumbent	BestBd	Gap	It/Node	Time
0	0	0.66177	0	310	-	0.66177	-	-	0s

0	0	1.00000	0	261	-	1.00000	-	-	0s
0	0	1.00000	0	257	-	1.00000	-	-	0s
0	0	2.00000	0	323	-	2.00000	-	-	0s
0	0	2.00000	0	314	-	2.00000	-	-	0s
0	0	2.00000	0	339	-	2.00000	-	-	0s
0	0	2.00000	0	317	-	2.00000	-	-	0s
0	0	2.00000	0	321	-	2.00000	-	-	0s
0	0	2.00000	0	342	-	2.00000	-	-	0s
0	0	2.00000	0	321	-	2.00000	-	-	1s
0	1	2.00000	0	321	-	2.00000	-	-	4s
767	539	4.00000	26	187	-	2.00000	-	97.9	5s
1281	829	4.00000	20	319	-	2.07253	-	99.3	10s
2312	1079	infeasible	33		-	3.00000	-	73.6	15s
4172	1379	infeasible	29		-	3.00000	-	104	20s
5809	2036	infeasible	57		-	3.00000	-	113	25s
7191	2610	infeasible	40		-	3.00000	-	120	30s
* 7228	2173		48		6.0000000	3.00000	50.0%	121	30s
H 7590	1579				5.0000000	3.00000	40.0%	122	31s

Cutting planes:

Cover: 1
 Implied bound: 2
 MIR: 62
 StrongCG: 2
 Flow cover: 75
 Zero half: 3
 RLT: 1
 Relax-and-lift: 2

Explored 9466 nodes (1131289 simplex iterations) in 34.38 seconds
 Thread count was 12 (of 12 available processors)

Solution count 2: 5 6

Optimal solution found (tolerance 1.00e-04)
 Best objective 5.000000000000e+00, best bound 5.000000000000e+00, gap 0.0000%

Let us extract the solution:

```
[1027]: solution = []
for t in range(max_moves):
    for i in range(18):
        if y[i, t].x > 0.5:
            solution += [moves_mapping[i]]
solution_alg = ' '.join(solution)
print(solution_alg)
```

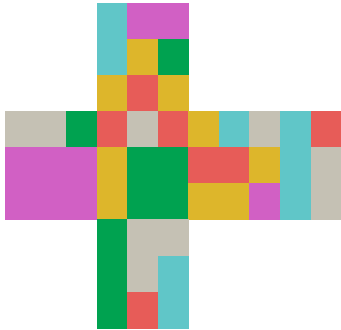
F R' F' L' B

Ok, so this was our original cube:

[1022]:

```
c
```

[1022]:



And now, we apply the obtain solution algorithm:

[1023]:

```
c(solution)
```

[1023]:



Indeed, the cube is solved! For sake of comparison, let us solve the cube with so-called *Corners-first method* that humans typically use to solve the cube:

[1024]:

```
from pycuber.solver import CFOPSSolver

solver = pc.solver.CFOPSSolver(fc)
solution = solver.solve(suppress_progress_messages=False)
```

Cross: U2 B' R B' D2 R D2

F2L('green', 'orange'): y R U' R' y' U F' U2 F

F2L('orange', 'blue'): y U R U2 R' U R U' R'

F2L('blue', 'red'): y R U' R' U F' U' F

F2L('red', 'green'): y U R U R' U' F' U' F

OLL: U2 R U R' U R U' R' U' R' F R F'

PLL: y U x' R U' R' D R U R' D' R U R' D R U' R' D'

FULL: U2 B' R B' D2 R D2 B U' B' U F' U2 F U B U2 B' U B U' B' L U' L' U B' U' B
U F U F' U' L' U' L U2 F U F' U F U' F' U' F' L F L' U R B' R' F R B R' F' R B
R' F R B' R' F'

```
[1025]: length_legacy_alg = sum([1 if not '2' in str(a) else 2 for a in solution])
length_legacy_alg
```

```
[1025]: 73
```

Hence, the suboptimal solution is much longer (but it is much faster to find it).

```
[ ]:
```