Rubik’s cube solver

FRS Project A.Y. 2020-2021

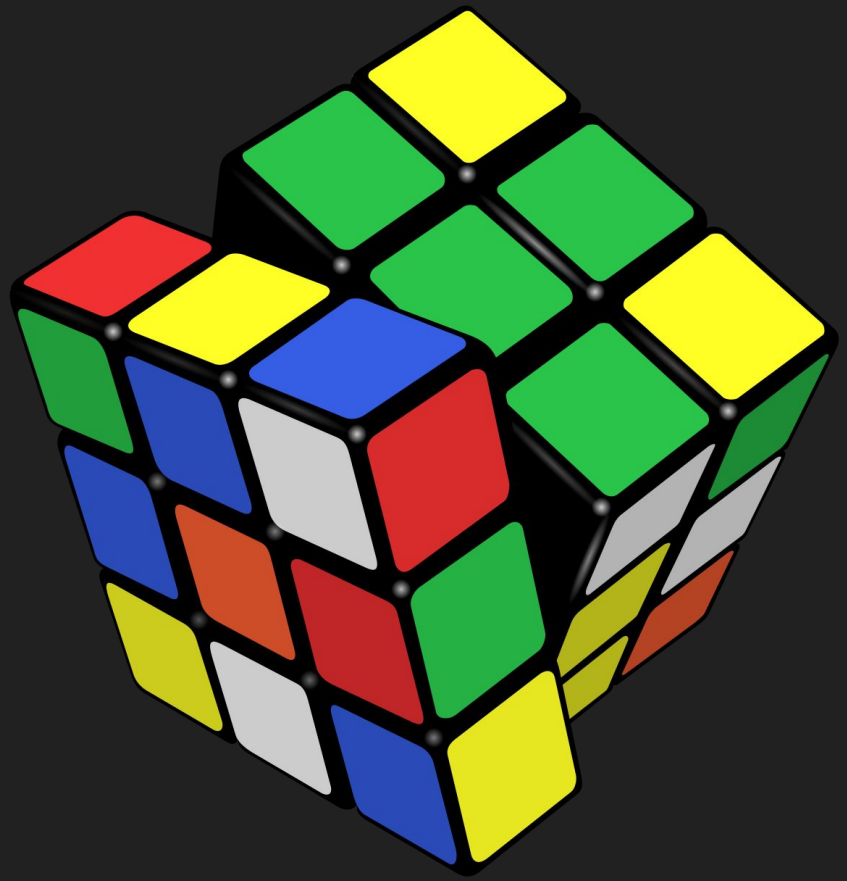
Tommaso Carletti
Dmitry Mingazov

An introduction

The **Rubik's Cube** is a 3-D combination puzzle invented in 1974 by Hungarian sculptor and professor of architecture Ernő Rubik.

It consists of a 3x3 rotating cube, with 12 possible moves from each perspective.

It is considered solved if every face contains only squares of the same color.



From domain to code

- Every face of the cube is represented by the following syntax:
  - face_name(C1 C2 C3 C4 C5 C6 C7 C8 C9) => C variables represent the colors inside the face
- The cube itself is composed by six faces, which in itself is called State:
  - s(face1 face2 face3 face4 face5 face6) => State
- The colors can assume any value, but canonical Rubik’s cube colors are used by default
- The 12 moves allowed from the cube (6 basic moves and the respective opposite ones) are implemented in the software, and they are called by the standard name
  - e.g. F : State -> State => this move performs a clockwise rotation of the Front face

The algorithm

The algorithm used to solve the Rubik’s cube can be broken down into 12 phases:

1. Forming a white cross
2. Forming a white face
3. Forming the first layer
4. Forming the second layer
5. Forming a yellow cross
6. Forming a yellow face
7. Swapping yellow corners
8. Swapping yellow edges
9. Looking at the cube
10. Realising it’s done already
11. ???
12. Profit

Algorithm implementation facets

- It uses more than 50 boolean operators that are used to guide the software through the resolution, reducing the allowed set of moves per state
- This makes for a greatly reduced number of rewritings and different states.
- The overall allowed transformations are the 12 basic moves, 10 combinations of these moves plus 3 rotations, which are enough to solve the entire cube.
- Lastly, there are 42 rewriting laws which uses the boolean operators and the transformation to actually solve the cube

Form the white cross

```
crl [toUpperEdgeResolvable] : S => U(S) if hasFrontWhiteEdge(S) and hasUpperWhiteEdge(S) .
crl [rotateWhiteDot] : S => yawCW(S) if isOnlyWhiteDotState(S) and not(hasFrontWhiteEdge(S)) .
crl [whiteDotLeft] : S => F(S) if hasWhiteEdgeInLeft(S) and not(hasUpperWhiteEdge(S)) .
crl [whiteDotRight] : S => Fi(S) if hasWhiteEdgeInRight(S) and not(hasUpperWhiteEdge(S)) .
crl [whiteDotBottom] : S => wDotB(S) if hasWhiteEdgeInFrontBottom(S) and not(hasUpperWhiteEdge(S)) .
--- whiteDotUp has one more condition to avoid possible loop with whiteDotBottom
crl [whiteDotUp] : S => F(F(S)) if hasWhiteEdgeInFrontUp(S) and not(hasUpperWhiteEdge(S)) and not(hasWhiteEdgeInFrontBottom(S)) .
```

The very first set of laws is used to create the white cross in the upper face. It looks for a white edge in the front face and once it finds one, it brings this edge to the upper face

Align the white cross

```
crl [toAlignedFront] : S => U(S) if isOnlyWhiteCrossState(S) .
crl [alignByFinalMove] : S => yCrossi(yawCW(yCross(yawCW(yawCW(S))))) if isAlignableByFinalMove(S) and isOnlyWhiteCrossState(S) .
crl [alignByFinalMoveInverse] : S => yCross(yawCCW(yCrossi(yawCW(yawCW(S))))) if isAlignableByInverseFinalMove(S) and isOnlyWhiteCrossState(S) .
crl [alignByLeft] : S => Ui(yCrossi(yawCW(yawCW(S)))) if isAlignableByLeft(S) and isOnlyWhiteCrossState(S) .
crl [alignByRight] : S => U(yCross(yawCW(yawCW(S)))) if isAlignableByRight(S) and isOnlyWhiteCrossState(S) .
crl [alignBySwap] : S => yCross(S) if isAlignableBySwap(S) and isOnlyWhiteCrossState(S) .
```

Once the white cross is created, it should be aligned with the center of the front, right, left and back faces. In order to do that, this set of laws perform some edges swappings on the upper face.

Form the white face

```
crl [toUpperRightCornerResolvable] : S => D(S) if isUpperRightWhiteCornerReallyResolvable(S) and isOnlyWhiteCrossAlignedState(S)
crl [rotateWhiteCross-solved] : S => yawCW(S) if isUpperRightCornerSolved(S) and isOnlyWhiteCrossAlignedState(S) .
crl [rotateWhiteCross-unresolvable] : S => yawCW(S) if (not(isUpperRightWhiteCornerReallyResolvable(S)) and not(isUpperRightCorne
crl [shiftDown] : S => D(S) if (not(isUpperRightWhiteCornerReallyResolvable(S))) and isUpperRightCornerWrongWhite(S) and isLowerR
crl [upperRightCornerWrong] : S => wCrossF(S) if isUpperRightCornerWrongWhite(S) and not(isLowerRightCornerWrongWhite(S)) and isO
crl [lowerRightCornerFront] : S => wCrossF(S) if isLowerRightWhiteCornerFront(S) and isOnlyWhiteCrossAlignedState(S) .
crl [lowerRightCornerRight] : S => wCrossR(S) if isLowerRightWhiteCornerRight(S) and isOnlyWhiteCrossAlignedState(S) .
crl [lowerRightCornerDown] : S => wCrossD(S) if isLowerRightWhiteCornerDown(S) and isOnlyWhiteCrossAlignedState(S) .
```

Once the white cross is formed, this set of laws look for all the remaining white corners in the cube and put them at the right upper face spot.

```
op wCrossF : State -> State .
eq wCrossF(S) = Fi(D(F(S))) .

op wCrossR : State -> State .
eq wCrossR(S) = R(Di(Ri(S))) .

op wCrossD : State -> State .
eq wCrossD(S) = F(Li(D(D(L(F(S)))))) .
```

Form the second layer

```
crl [toFrontResolvable] : S => U(S) if isOnlyFrontResolvable(S) .
crl [firstLayer] : S => fLayer(S) if isFrontRightResolvable(S) .
crl [firstLayerMissplaced] : S => fLayer(S) if isFrontRightMissplaced(S) .
crl [firstLayerInverse] : S => fLayeri(S) if isFrontLeftResolvable(S) or isFrontLeftMissplaced(S) .
crl [firstLayerMissplacedInverse] : S => fLayeri(S) if isFrontLeftMissplaced(S) .
crl [rotateFirstLayer] : S => yawCW(S) if mustRotate(S) .
```

Once the white face is formed, the cube is rolled twice forward and begins the second part of the algorithm. In particular, this set of laws looks for every edge suitable to be placed in the front right or front left position. Once it finds it, the algorithm will put it into place.

```
op fLayer : State -> State .
eq fLayer(S) = F(U(Fi(Ui(Ri(Ui(R(U(S))))))) .
```

```
op fLayeri : State -> State .
eq fLayeri(S) = Fi(Ui(F(U(L(U(Li(Ui(S))))))) .
```

Form the yellow cross

```
crl [rotateSecondLayer] : S => yawCW(S) if isOnlySecondLayerState(S) .
crl [secondLayer] : S => sLayer(S) if isOnlySecondLayerState(S) .
crl [secondLayerInverse] : S => sLayeri(S) if isOnlySecondLayerState(S) .
```

This set of laws allows the algorithm to form a yellow cross in the upper face, using the proper sets of moves.

```
op sLayer : State -> State .
eq sLayer(S) = R(F(U(Fi(Ui(Ri(S)))))) .

op sLayeri : State -> State .
eq sLayeri(S) = Li(Fi(Ui(F(U(L(S)))))) .
```

Form the yellow face

```
crl [rotateYellowCross] : S => yawCW(S) if isOnlyYellowCrossState(S) .
crl [yellowCross] : S => yCross(S) if isOnlyYellowCrossState(S) .
crl [yellowCrossInverse] : S => yCrossi(S) if isOnlyYellowCrossState(S) .
```

Once the yellow cross is formed, the set **yCross** and its inverse will be executed until a yellow face will be formed.

```
op yCross : State -> State .
eq yCross(S) = Ri(U(U(R(U(Ri(U(R(S))))))) .

op yCrossi : State -> State .
eq yCrossi(S) = L(Ui(Ui(Li(Ui(L(Ui(Li(S))))))) .
```

These movesets will just swap yellow edges preserving the cross state.

Swapping yellow corners

```
crl [rotateYellowFace] : S => yawCW(S) if isOnlyYellowFaceState(S) .
crl [yellowFace] : S => yFace(S) if isOnlyYellowFaceState(S) .
crl [yellowFaceInverse] : S => yFacei(S) if isOnlyYellowFaceState(S) .
```

Once the yellow face is formed, the set **yFace** and its inverse will swap yellow corners until the cube state ready to be immediately solved appears.

```
op yFace : State -> State .
eq yFace(S) = R(R(B(B(Ri(Fi(R(B(B(Ri(F(Ri(S))))))))))) .

op yFacei : State -> State .
eq yFacei(S) = Li(Li(Bi(Bi(L(F(Li(Bi(Bi((L(Fi(L(S))))))))))))) .
```

Swapping yellow edges

```
crl [rotate] : S => yawCW(S) if isReallyFinalState(S) and not(isFinalState(S)) .
crl [finalMove] : S => yCrossi(yawCW(yCross(S))) if isFinalState(S) .
crl [finalMoveInverse] : S => yCross(yawCCW(yCrossi(S))) if isFinalState(S) .
```

Once the yellow face is complete, the edges of the yellow face might not be aligned with the right lateral face. The **finalMove** and **finalMoveInverse** rewriting laws tackle the issue by swapping front and right and front and left topmost edges respectively. **rotate** only comes into play if an edge can be eventually solved but can’t be right away with either rewriting rule.

```
op yCross : State -> State .
eq yCross(S) = Ri(U(U(R(U(Ri(U(R(S)))))))) .
```

```
op yCrossi : State -> State .
eq yCrossi(S) = L(Ui(Ui(Li(Ui(L(Ui(Li(S)))))))) .
```

Shortcomings

Due to the algorithm used and the nature of Maude alike, the software could use a few improvements:

- The algorithm isn’t by large the most efficient, as it simulates how normal 22 years old CS students would solve a Rubik’s cube.
- Due to Maude generating the entire tree while looking for a solution using the search interpreter command, solving the cube from a generic state takes too long: this led to executing the solution by looking for intermediate states and feeding these newly-found states to the interpreter to find more advanced states until the cube is solved.

Example execution: execution chunks

```
dmitry@DESKTOP-HBR1611:~/rubik-solver$ ../maude-3.1/maude.linux64 rubik.maude
                     \||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||\
            Maude 3.1 built: Oct 12 2020 20:12:31
            Copyright 1997-2020 SRI International
                  Tue Jan 19 01:34:04 2021
Maude> search[1] in RUBIK-WC : scrambleState =>* S such that isWhiteCrossState(S) .
search [1] in RUBIK-WC : scrambleState =>* S such that isWhiteCrossState(S) = true .

Solution 1 (state 9)
states: 10  rewrites: 744 in 0ms cpu (0ms real) (~ rewrites/second)
S --> s(left(B B Y R R O W Y G) front(B G R Y B Y W B G) right(G O B G O R Y O W) back(Y R W B G Y G R R) up(O W O W W W R W Y) down(R O O B Y G B G O))
Maude> search[1] in RUBIK-ALG : s(left(B B Y R R O W Y G) front(B G R Y B Y W B G) right(G O B G O R Y O W) back(Y R W B G Y G R R) up(O W O W W W R W Y) down(R O O B Y G B G O)) =>* S such that isWhiteFaceState(S) .
search [1] in RUBIK-ALG : s(left(B B Y R R O W Y G) front(B G R Y B Y W B G) right(G O B G O R Y O W) back(Y R W B G Y G R R) up(O W O W W W R W Y) down(R O O B Y G B G O)) =>* S such that isWhiteFaceState(S) = true .

Solution 1 (state 3450)
states: 3451  rewrites: 3237215 in 0ms cpu (717ms real) (~ rewrites/second)
S --> s(left(G G G O G Y R R G) front(R R R B R R Y O R) right(B B B G B Y B O Y) back(O O O G O B B Y G) up(W W W W W W W W W) down(O G Y B Y Y Y R O))
Maude> search[1] in RUBIK-ALG : s(left(G G G O G Y R R G) front(R R R B R R Y O R) right(B B B G B Y B O Y) back(O O O G O B B Y G) up(W W W W W W W W W) down(O G Y B Y Y Y R O)) =>* S such that isSolved(S) = true .
search [1] in RUBIK-ALG : s(left(G G G O G Y R R G) front(R R R B R R Y O R) right(B B B G B Y B O Y) back(O O O G O B B Y G) up(W W W W W W W W W) down(O G Y B Y Y Y R O)) =>* S such that isSolved(S) = true .

Solution 1 (state 12390)
states: 12391  rewrites: 9484724 in 0ms cpu (2536ms real) (~ rewrites/second)
S --> s(left(O O O O O O O O O) front(B B B B B B B B B) right(R R R R R R R R R) back(G G G G G G G G G) up(Y Y Y Y Y Y Y Y Y) down(W W W W W W W W W))
Maude>
```

Example execution: successful path

```
Solution 1 (state 12390)
states: 12391  rewrites: 9484724 in 0ms cpu (2536ms real) (~ rewrites/second)
S --> s(left(O O O O O O O O O) front(B B B B B B B B B) right(R R R R R R R R R) back(G G G G G G G G G) up(Y Y Y Y Y Y Y Y Y) down(W W W W W W W W W))
Maude> show path 12390 .
state 0, State: s(left(G G G O G Y R R G) front(R R R B R R Y O R) right(B B B G B Y B O Y) back(O O O G O B B Y G) up(W W W W W W W W W) down(O G Y B Y Y Y R O))
===[ crl S => roll(roll(S)) if isOnlyWhiteFaceState(S) = true [label rollTheCube] . ]===>
state 1, State: s(left(G R R Y G O G G G) front(G Y B B O G O O O) right(Y O B Y B G B B B) back(R O Y R R B R R R) up(O G Y B Y Y Y R O) down(W W W W W W W W W))
===[ crl S => fLayeri(S) if isFrontLeftMissplaced(S) or isFrontLeftResolvable(S) = true [label firstLayerInverse] . ]===>
state 5, State: s(left(R O R Y G R G G G) front(G O O Y O G O O O) right(Y B Y Y B G B B B) back(O O B R R B R R R) up(Y B B G Y R Y Y G) down(W W W W W W W W W))
===[ crl S => U(S) if isOnlyFrontResolvable(S) = true [label toFrontResolvable] . ]===>
state 12, State: s(left(G O O Y G R G G G) front(Y B Y Y O G O O O) right(O O B Y B G B B B) back(R O R R R B R R R) up(Y G Y Y Y B G R B) down(W W W W W W W W W))
===[ crl S => U(S) if isOnlyFrontResolvable(S) = true [label toFrontResolvable] . ]===>
state 25, State: s(left(Y B Y Y G R G G G) front(O O B Y O G O O O) right(R O R Y B G B B B) back(G O O R R B R R R) up(G Y Y R Y G B B Y) down(W W W W W W W W W))
===[ crl S => U(S) if isOnlyFrontResolvable(S) = true [label toFrontResolvable] . ]===>
state 45, State: s(left(O O B Y G R G G G) front(R O R Y O G O O O) right(G O O Y B G B B B) back(Y B Y R R B R R R) up(B R G B Y Y Y G Y) down(W W W W W W W W W))
===[ crl S => fLayeri(S) if isFrontLeftMissplaced(S) or isFrontLeftResolvable(S) = true [label firstLayerInverse] . ]===>
state 84, State: s(left(Y B B Y G G G G G) front(R O B O O G O O O) right(Y B G Y B G B B B) back(Y R O R R B R R R) up(G Y R R Y O Y Y O) down(W W W W W W W W W))
===[ crl S => yawCW(S) if mustRotate(S) = true [label rotateFirstLayer] . ]===>
state 152, State: s(left(R O B O O G O O O) front(Y B G Y B G B B B) right(Y R O R R B R R R) back(Y B B Y G G G G G) up(Y R G Y Y Y O O R) down(W W W W W W W W W))
===[ crl S => fLayeri(S) if isFrontLeftMissplaced(S) or isFrontLeftResolvable(S) = true [label firstLayerInverse] . ]===>
state 272, State: s(left(Y B B O O O O O O) front(Y R Y B B G B B B) right(B Y Y R B R R R R) back(R G O Y G G G G G) up(G Y G R Y O O Y R) down(W W W W W W W W W))
===[ crl S => fLayer(S) if isFrontRightMissplaced(S) = true [label firstLayerMissplaced] . ]===>
state 472, State: s(left(B O R O O O O O O) front(G B Y B B R B B B) right(B G O Y R B R R R) back(Y R O Y G G G G G) up(Y G G Y Y Y Y R R) down(W W W W W W W W W))
===[ crl S => fLayer(S) if isFrontRightResolvable(S) = true [label firstLayer] . ]===>
state 785, State: s(left(R Y Y O O O O O O) front(G O Y B B B B B B) right(B R O R R B R R R) back(B Y Y Y G G G G G) up(G R Y G Y G O Y R) down(W W W W W W W W W))
===[ crl S => yawCW(S) if mustRotate(S) = true [label rotateFirstLayer] . ]===>
state 1206, State: s(left(G O Y B B B B B B) front(B R O R R B R R R) right(B Y Y Y G G G G G) back(R Y Y O O O O O O) up(O G G Y Y R R G Y) down(W W W W W W W W W))
===[ crl S => fLayer(S) if isFrontRightResolvable(S) = true [label firstLayer] . ]===>
state 1742, State: s(left(Y R R B B B B B B) front(G O O R R R R R R) right(B Y Y G G G G G G) back(G Y R O O O O O O) up(B B O Y Y G Y Y Y) down(W W W W W W W W W))
===[ crl S => yawCW(S) if isOnlySecondLayerState(S) = true [label rotateSecondLayer] . ]===>
state 2360, State: s(left(G O O R R R R R R) front(B Y Y G G G G G G) right(G Y R O O O O O O) back(Y R R B B B B B B) up(Y Y B Y Y B Y G O) down(W W W W W W W W W))
===[ crl S => sLayer(S) if isOnlySecondLayerState(S) = true [label secondLayer] . ]===>
state 3031, State: s(left(O B R R R R R R R) front(G O R G G G G G G) right(Y G G O O O O O O) back(O R Y B B B B B B) up(B Y Y Y Y Y Y Y B) down(W W W W W W W W W))
===[ crl S => yCrossi(S) if isOnlyYellowCrossState(S) = true [label yellowCrossInverse] . ]===>
state 3790, State: s(left(Y G Y R R R R R R) front(G O B G G G G G G) right(Y R Y O O O O O O) back(R B R B B B B B B) up(B Y G Y Y Y O Y O) down(W W W W W W W W W))
===[ crl S => Ui(yCrossi(yawCW(yawCW(S)))) if isAlignableByLeft(S) and isOnlyWhiteCrossState(S) = true [label alignByLeft] . ]===>
state 4643, State: s(left(Y R R O O O O O O) front(Y G O B B B B B B) right(Y B O R R R R R R) back(B O G G G G G G G) up(R Y Y Y Y Y B Y G) down(W W W W W W W W W))
===[ crl S => U(S) if isOnlyWhiteCrossState(S) = true [label toAlignedFront] . ]===>
state 5754, State: s(left(Y G O O O O O O O) front(Y B O B B B B B B) right(B O G R R R R R R) back(Y R R G G G G G G) up(B Y R Y Y Y G Y Y) down(W W W W W W W W W))
===[ crl S => yCrossi(S) if isOnlyYellowCrossState(S) = true [label yellowCrossInverse] . ]===>
state 7718, State: s(left(B O R O O O O O O) front(G B B B B B B B B) right(R R G R R R R R R) back(O G O G G G G G G) up(Y Y Y Y Y Y Y Y Y) down(W W W W W W W W W))
===[ crl S => yFacei(S) if isOnlyYellowFaceState(S) = true [label yellowFaceInverse] . ]===>
state 12390, State: s(left(O O O O O O O O O) front(B B B B B B B B B) right(R R R R R R R R R) back(G G G G G G G G G) up(Y Y Y Y Y Y Y Y Y) down(W W W W W W W W W))
Maude>
```

Demo

No really, stop looking at these slides and pay attention to Dmitry

Thanks everyone for the attention

Any questions?