# cubesProject Summary

Solving the popular kid's toy that asks you to place coloured cubes in a number of slots so that now side has the same colours.

# Propositions

* Marking note: This section would be complete and given full points.

We used several types of propositions to model the colour cubes problem. We list them all here, and do so in first-order logic form (since the propositions were created as Bauhaus objects). The options for each of the dice, sides, etc. were as follows:

DICE = [1, 2, 3, 4]  
SLOTS = [1, 2, 3, 4]  
SIDE = [1, 2, 3, 4, 5, 6]  
COLOURS = ['red', 'green', 'blue', 'yellow']  
DIRECTION = ['top', 'bottom', 'left', 'right', 'front', 'back']

* **DiceSideCol(d,s,c)**: Dice **d** has colour **c** on side **s**.
* **DiceSideDir(d,s,r)**: Dice **d** has side **s** pointing in direction **r**.
* **DiceInSlot(d,s)**: Dice **d** is in slot **s**.
* **BoxColour(s,d,c)**: Slot **s** has colour **c** pointing in direction **d**. Note, ‘left’ and ‘right’ directions were not used for this proposition.

# Constraints

* Marking note: The existing (complete) constraints are good, but the others listed are missing.

Several types of constraints were used, and the vast majority of them were specified with properly constructed for-loops. We group them into three broad categories: (1) directions & colours; (2) slot constraints; and (3) game rules.

## Directions & Colours

* Each dice must have the side appear in exactly one direction
* Each direction must have exactly one side
* *Each dice side must have exactly one colour*
  + For every side **s** and dice **d**, exactly one colour is selected:  
    exactly-one(DiceSideCol(d,s,’red’), DiceSideCol(d,s,’green’),  
     DiceSideCol(d,s,’blue’), DiceSideCol(d,s,’yellow’))
* Opposite sides must add up to 7
* 1, 2, 3 must be clockwise around a corner
* Specific dice configuration

## Slot Constraints

* A dice can only be in one slot
* *If a dice of a particular configuration is in a slot, then the colour facing each direction is defined*
  + The following constraint is added for every dice **d**, slot **l**, side **s**, colour **c**, and direction (excluding ‘right’ and ‘left’) **r**:  
      
     (DiceSideCol(d,s,c) DiceSideDir(d,s,r) DiceInSlot(d,l)) BoxColour(l,r,c)
* A box side colour must have exactly one colour

## Game Rules

With all of the previous constraints in place, there is only a single type of constraint needed to encode the game’s rules: *For all of the sides (other than left and right), the box colours are unique.*

# Model Exploration

* Marking note: Good start with this, but we’d expect at least 4 or so more subsections that explore things here.

## Dice Details

At one point, I wasn’t sure what was happening with the dice placed in a slot forcing the case colours to be set correctly. In order to debug this, I wound up implementing a function that printed all of the information about the propositions of a certain dice that happened to be true. This function looked like this:

Text

Description automatically generated

Essentially, we just want to find all of the parts of the solution that deal with some small part of the overall problem. We were able to use this to figure out what was happening with a particular dice, and cross reference it with the visualization used for the entire case.

## Starting with just one dice

## Using model counts to confirm orientations

## Picking slots to simplify problem

## Measuring difficulty as # of solutions to cubes

## Randomly configuring cube colours to find easy/hard instances

## Greedy colouring of cubes to find really constrained instances

# Jape Proof Ideas

Marking note: 3 great ideas, and 2 perfectly complete proofs. 2/3

We will work on a simplified setting for these proofs. For example, fewer cubes, colours, slots, etc.

## All-but-one colour implies the last

***For a side (e.g., “front”), having all but one colour implies that the last colour will be in the final slot.***

Here, we use P<s><c> to represent that “the front side in slot <s> is colour <c>”. Our premises include the following:

* Table

  Description automatically generatedColour can’t appear in two slots:  
  ¬P1R ∨ ¬P2R,  
  ¬P1G ∨ ¬P2G,  
  …
* Every slot has some colour:  
  P1R ∨ P1G ∨ P1B,

P2R ∨ P2G ∨ P2B,

P3R ∨ P3G ∨ P3B,

* First slot is red and second is green:  
  P1R,

P2G

Finally, we want to deduce the final slot colour, P3B. The complete jape proof is shown on the right.

## Can’t fill 3 slots with 2 colours

If we only have two colours modeled, then there’s no way to satisfy things for 3 slots.

This is a long proof of disjunctive case-based reasoning. We use the same propositions as before, except remove any mention of the colour blue. The sequent to prove is…  
¬P1R ∨ ¬P2R, ¬P1G ∨ ¬P2G, ¬P1R ∨ ¬P3R, ¬P1G ∨ ¬P3G, ¬P2R ∨ ¬P3R, ¬P2G ∨ ¬P3G, P1R ∨ P1G, P2R ∨ P2G, P3R ∨ P3G ⊢ ⊥

…and the proof is as follows:

`Table

Description automatically generated

## All-red dice implies blue beside

***If there are just two slots, and an all-red dice is placed in the first slot, then the second must be blue.***

{{{ proof left as an exercise for the reader 😉. **NOTE:** you are not allowed to do the same for your project – doing so (like I have) would mean losing all marks for the final sequent }}

# First-Order Extension

If we were to extend this to first-order logic, then the quantification would mirror the code quite closely – iterating over the cubes, colours, sides, etc. This naturally extends things to an arbitrary number of colours, cubes, etc. An example of some of the constraints in a first-order extension:

* *If a dice of a particular configuration is in a slot, then the colour facing each direction is defined*   
  d.s.c.r.l. ( (Dice(d) Side(s) Colour(c) Direction(r) Slot(l))   
   ( (DiceSideCol(d,s,c) DiceSideDir(d,s,r) DiceInSlot(d,l)) BoxColour(l,r,c) ) )
* One aspect of the model that couldn’t easily be generalized is the precise dice colouring – this requires constraints specific to a certain object (i.e., the particular dice), and the colours on each side of it.