DSL solution to the ARC challenge

Johan Sokrates Wind

June 2020

1 Contestant

Competition name: Abstraction and Reasoning Challenge

Team name: icecuber
Private leaderboard score: 0.794
Private leaderboard place: 1st

Author: Johan Sokrates Wind
Location: Tromsø, Norway
Email: *top-quarks@protonmail.com*

2 About the author

I hold a master's degree in Industrial Mathematics from Norwegian University of Science and Technology. However, I believe my experience with competitive programming (CP) was more useful for this contest. CP has trained me to quickly and efficiently write simple algorithms such as flood fill, dynamic programming and Huffman coding. Since a large part of my solution was generalizing and implementing image transformations, and run-time performance was important, this fit my skills well. Larger projects I've done (like implementing visual odometry on a mobile phone, and the TrackML Kaggle contest) were also useful, as they helped me structure my code in a way that allows adding new ideas easily.

The aim of measuring AGI intrigued me into trying the competition. The format also fit me well, because I could write an efficient solution from scratch, and there were no well established approaches to solve it.

I worked on and off from the start of the competition. If I had to guess, I would estimate a bit over a month of full time work went into it in total.

3 Summary of approach

The main component of my solution is a DSL which applies up to 4 of 142 unary transformations (based on 42 different functions, where some have multiple vari-

ants). I enumerate the transformations efficiently by reducing duplicates, and then combine them by greedily stacking them to fit training samples. Everything is implemented efficiently in C++ (with no dependencies) and running in parallel. A simple scheduler tries to use the 9 hour / 16 GB memory budget fully.

4 Transformation selection / engineering

The most important image transformations:

- Cut (image) → list of images
  Tries to figure out a background color and splits the remaining pixels into corner connected groups.
- filterCol (image, color) → image
  Erases all colors except the given one (sets them to 0).
- colShape (image, color) → image
  Change all non-zero pixels to "color".
- composeGrowing (list of images) → image
  Stack the list of images on top of each other, treating 0 as transparent. The image with the fewest non-zero pixels is at the top.
- compress (image) → image
  Extract minimal sub-image containing all non-zero pixels.
- rigid (image, id) → image
  Perform rotation (0/90/180/270 degrees) and/or flip.
- pickMax (list of images, id) → image
  Extract the image with maximum property, for example id = 0 extracts the image with the most non-zero pixels.

I constructed transformations by hand-solving 100 training tasks and 100 evaluation tasks, and then extracting useful functions. Generalizing when it seemed reasonable (like adding all rotations, if I used one of them). I didn't try to prune the transformations, since the given tasks did not seem representative of the tasks needed on the leaderboard.

The transformations stacked very well, even solving several tasks in which I used other transformations (not available to the model) during hand-solving.

5 Ensembling

In the final model I run 4 different configurations and ensemble the predictions. I run transformations search depth 3, depth 3 augmented with diagonal flips (times two diagonal flips), and finally run depth 4 until I run out of time or

memory. The best predictions are picked according to the following criteria, with the top criterion being the most important one:

- Solved the most training samples
- Least depth solution
- Least stacked images in the greedy stacker

6 Tricks

Augmenting my samples with diagonally flipped tasks, was a simple trick which gave me significantly better score. Preprocessing all samples by remapping colors according to some heuristics, also worked surprisingly well.

I believe my main advantage over most other competitors was my experience from competitive programming. It allowed me to quickly and efficiently write large amounts of image transformations in C++, which let me search through many more combinations of transformations compared to a python implementation or an otherwise less optimized solution.

7 Execution time

A natural way to make my approach run faster is to reduce search depth. When I use the full 9 hours I can run about half the problems at depth 4, while running at depth 3 is about 20× faster (and takes 20× less memory). During development I would run at depth 2, which is again about 15× faster than depth 3, while solving about 80% as many tasks on the evaluation set.

8 Code

The implementation is available at https://github.com/top-quarks/ARC-solution