CS 7637: Project 1 Journal

Zachary Todd
ztodd3@gatech.edu

Abstract—This paper will outline the approaches I took to create a knowledge-based AI agent capable of solving Raven's Progressive Matrices.

1 INTRODUCTION

This paper outlines my approach and reflections for solving Raven's Progressive Matrices using knowledge-based AI (KBAI). At first glance, it appears as if one could solve these types of problems by creating a long list of if/else statements that would capture many different categories of patterns that a problem set might consist of. This approach would be limited to the amount of patterns I could think of and would not be an approach using KBAI. I have decided to use Java to solve this problem as I am most familiar with Python and would like to broaden my programming experience.

1.1 Initial approach

To solve these Raven's Problems, I will be implementing an agent which uses multiple approaches. Each of these approaches will check the given possible solutions compared to the given problem and assign a percentage of certainty to the answer. The agent will then choose the solution with the highest cumulative percentage. In the event that an approach finds one solution with 100% certainty, it will immediately choose that solution. Additionally, I will keep track of any method of transformation that matches for all rows or columns even if no transformation is found for both.

Simple—The first approach will be simple checks for rotation, translation, scaling, and mirroring of the entire image. This will be relatively quick to check and will be sufficient to answer some of the more basic problems.

Shapes—The second method for solving will consist of finding shapes within the image, identifying if those same shapes appear in other images, and seeing if there is a consistent transformation occurring between images.

Repetition—A third approach will check for patterns of repetition between images. For instance, a sequence of circle, star, square, and triangle being iterated over. There is no transformation occurring between images, but shapes are being replaced in a predictable pattern.

Duplication/deletion—The next approach will check for duplications and deletions of shapes. This will detect shapes in images and check if those shapes occur more or less times in other images.

Comparison—This method will attempt to pair distinct shapes between images and see if a pattern exists for those pairings. This could identify if everything in an image stays the same, except the shape at the very center of the image becomes smaller and smaller.

Combination—The final check will consist of a combination of the other 4 approaches. This check will get **double** weight when it assigns a percentage of confidence to an answer as it will be able to detect patterns over the widest range of possible inputs. This check will be the most difficult to implement and will take the longest to compute.

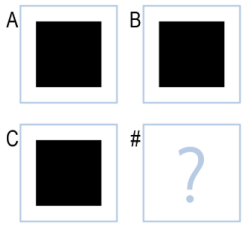


Figure 1—For clarity, I will refer to different Images in a 2x2 Raven's Problem as A, B, C, and # as shown above for the rest of this paper.

2 EARLY BLUNDERS (05/24)

My submission 1 and 2 resulted in errors. I initially got a build error due to my use of an import which was not accepted by the autograder. I removed this import

statement (as I was not actually using the imported package anymore) and received an execution error. I found the root cause of the error to be the GUI functions I was using for debugging. I again removed the code that was causing the error and submitted my code to the error check. There were no issues, so I submitted my project a third time and saw a successful run. In order to avoid wasting submissions like this in the future, I will always run my code against the error check before submitting to the autograder.

3 FORWARD PROGRESS (05/24) #3

My first *real* submission was submission 3 and contained code for the **Simple** check algorithm applied to 2x2 problems. Within this algorithm, I created an image comparison function which went pixel by pixel and had an error threshold of 5%. I used a threshold instead of looking for an exact match because guesses I generated tended to be very close to the right answer, but not exact (see Figure 2). I tried various thresholds from 0-10% and found that 5% approved similar images without accepting images that were not at all the same.

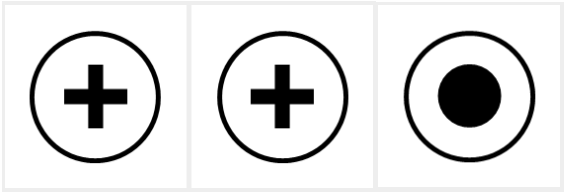


Figure 2—Image created by algorithm (left) and image given as a possible answer (center). These two images look identical to a human observer, but a pixel by pixel comparison reveals a 4.48% difference. Another image given as a possible answer (right) is easily distinguishable from the algorithm's guess but is only 9.39% different.

This first implementation of the Simple check included exact match, rotations for 90, 180, and 270 degrees, and vertical/horizontal mirroring. This was done by taking image A and performing various transformations on it then comparing it to image B and image C. If the transformed image matched, I attempted to perform the row transformation on the image C or the column transformation to image B. This resulted in a guess. Finally, I compared the guess to each of the given possible

answers. If one matched the guess, the answer had been found. If I was unable to find a match, I kept track of all the found horizontal and vertical transformations.

3.1 Performance

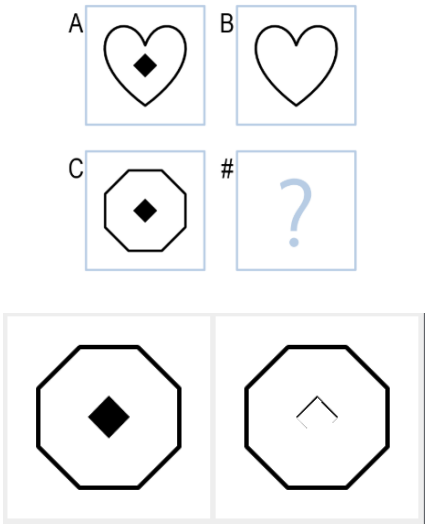
This first submission performed better than expected. It answered 17/24 correctly from the basic and test problems and 8/24 correctly from the Raven's and challenge problems. It did not guess incorrectly for the basic or test problems and guessed incorrectly 4 times in the Raven's and challenge problems. The algorithm is fairly efficient in terms of speed but needs to improve in accuracy.

The way in which Simple check approaches problems is **similar** to how a human would approach these questions. It takes a sort of trial and error approach by thinking of different transformations that exist in the rows and columns. It then makes a guess and sees if its guess was correct. Unfortunately, this implementation only applies to the *whole* image. Humans can distinguish the different shapes within an image and see if transformations are occurring on these smaller pieces of the image.

This implementation was **not** able to recognize shapes being filled in, deleted, added, combinations of transformations, or scaling and performed poorly on questions that had these elements. Furthermore, it performed poorly on any question that involved transformations of subsections of an image.

4 SECOND TRY (06/02) #4

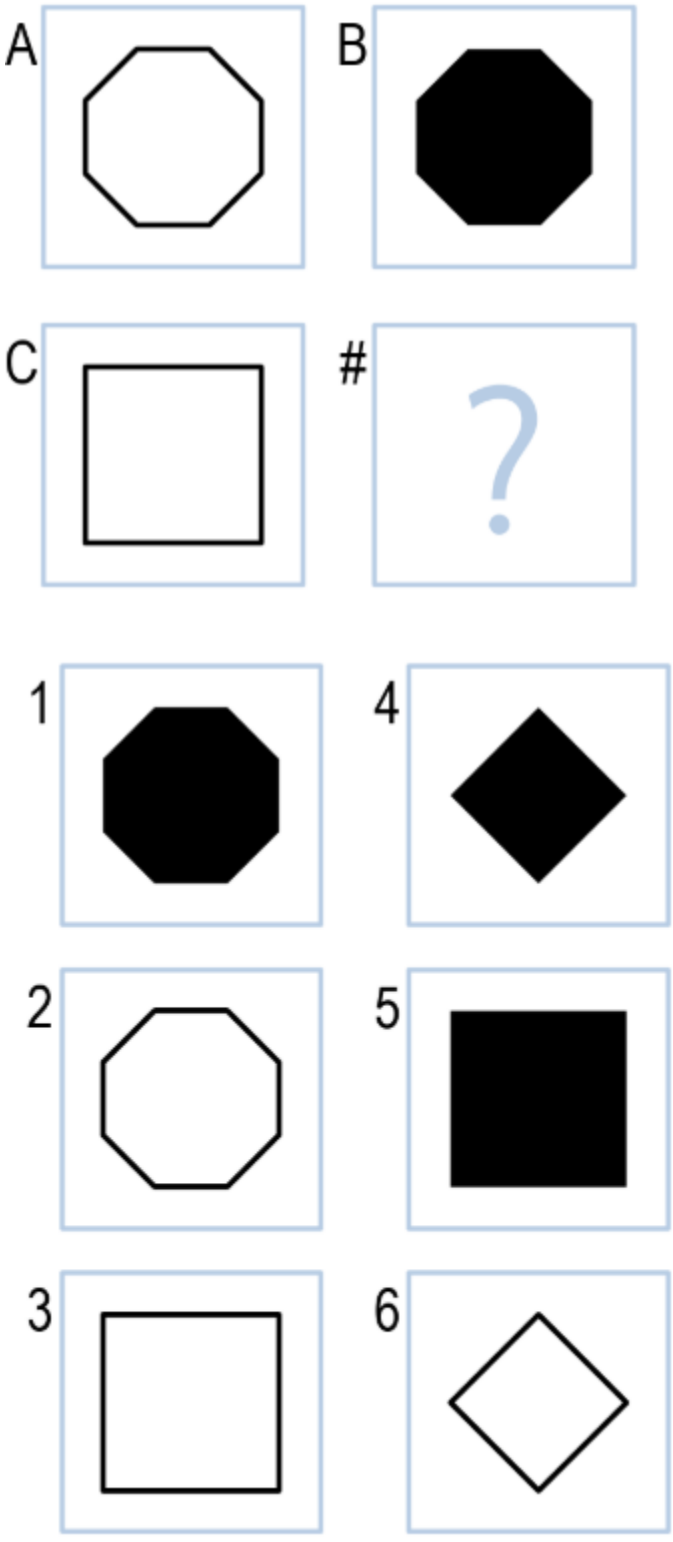
The second real submission (number 4), contained code for checking if pixels had been added or deleted from an image. This was done with a pixel-by-pixel comparison of one image to the next. I kept track of the sum of pixels added and pixels removed. I then took the difference of these sums and if the percentage of the difference compared to the whole image was greater than my error threshold of 5%, I would attempt to create a candidate solution by adding/removing the same pixels from the problem.



5 RESIDUE REMOVAL (06/04) #5

Removing residue from previous submission as well as adding a checker for both horizontal and vertical transforms combined. I attempted to make a function that would shift an image if it was slightly off from a possible answer, but was unable to make it work. I also kept track of the highest percentage for a certain answer across all possible transforms. Before, I was returning an answer as soon as I found something with over 95% confidence.

It answered 20/24 correctly from the basic and test problems and 13/24 correctly from the Raven's and challenge problems. My agent guessed incorrectly for the remaining 4 basic and test problems and 9 times from the Raven's and challenge problems. The algorithm is has slowed down considerably, and is more apt to guess, but is still very quick.


A
B
C
#
?
1
4
2
5
3
6