CS 4635 - Knowledge Based AI
Brad Ware
Project 1 Design Report

Contents

1. How does it work?

A. Strategy

B. Architecture

2. Application

A. Example Problem Walk-through

3. Shortcomings

A. When does it fall short?

B. Why?

4. Potential for improvements

A. What could have been done better?

1. How does it work?

There were many ways to go about designing this agent to solve Raven's progressive matrices, but one stood out to me above the rest. I could have designed a simple heuristic based on the changes that needed to be made to figure C to pick the answer choice that reflected the least change. However, I felt that this strategy may be inadequate for more complex problems that require many changes to figure C. So to combat this, I decided to try and match the same changes from figure A to B when choosing a pair for C. This strategy would at least allow the most similar match for C based on the information given.

A. Strategy

From a very high-level perspective, the strategy is to analyze the differences between figures A and B, and then choose an answer choice that reflected the most similar differences when compared to figure C. By using this strategy, I would not just find relationships between two figures, but would find the **key** characteristics that would determine which figure would be the right match for C. Then based on those characteristics, the agent would try to pick an answer that best resembled a duplicate of the relationship between figures A and B. Here is an overview of what the agent is computing when the function solve is being run:

The agent stores figures A and B in a separate wrapper object that gives a score to each figure's object based on its similarity. It then stores each object (Z, Y, X…) in a hash table with the object name as the key and it's score as the value.

This same step is repeated for each answer choice and is then stored in a separate wrapper object with figure C.

After that, comparisons are made between the hash maps of C and an answer choice, and the one of figures A and B. If the score is the same for an object, then the answer choice's score is incremented. The figure with the highest score is then returned as the answer.

However, because ties can occur frequently, a list is used to keep track of all the answers that tied with the largest score.

If the list is greater than 1, then each answer is further analyzed by comparing object Z of the answer figure and C's object Z. That heuristic applies a score based on the least amount of changes that pair has compared to figure A and B, the given pair.

Finally, the answer choice with the least amount of differences between its Z object and C's, compared to the object pair of figures A and B is returned as the answer.

B. Architecture

To implement this strategy, two new helper classes were defined to contain the relationships between objects and figures. The first is Pair, which needs two RavensFigures as parameters, and the second is ObjectPair, which needs two RavensObject as parameters.

Class Pair (Ravens Figure, Ravens Figure)

The purpose of this class was to track the relationship differences between each of the objects in the different figures. This class kept a hash map which stored numerical values based on the similarities between each object with the same name (Z, Y, X…).

Variables

- *fig1*: the first RavensFigure to be compared
- *fig2*: the second RavensFigure to be compared
- *fig1Objects*: An array list containing all of fig1's RavensObjects
- *fig2Objects*: An array list containing all of fig2's RavensObjects
- *numObjectsDiff*: a boolean to keep track if the two figures contained the same number of objects
- *totalScore*: the score of the relationship between the two figures
- *c*: character to keep track of each object name
- *asciivalue*: integer value associated with each character
- *figValues*: a hash map containing each object name as the key and the score associated as its value.

Key Functions

- *calcfigValues: compares each object of the two figures and scores them based on their similarities between their shape, size, fill, and angle. It then stores the name associated with the objects being compared and their score.*
- *getfigValues: returns the hash map constructed through calcFigValues*

Class ObjectPair (Ravens Object, Ravens Object)

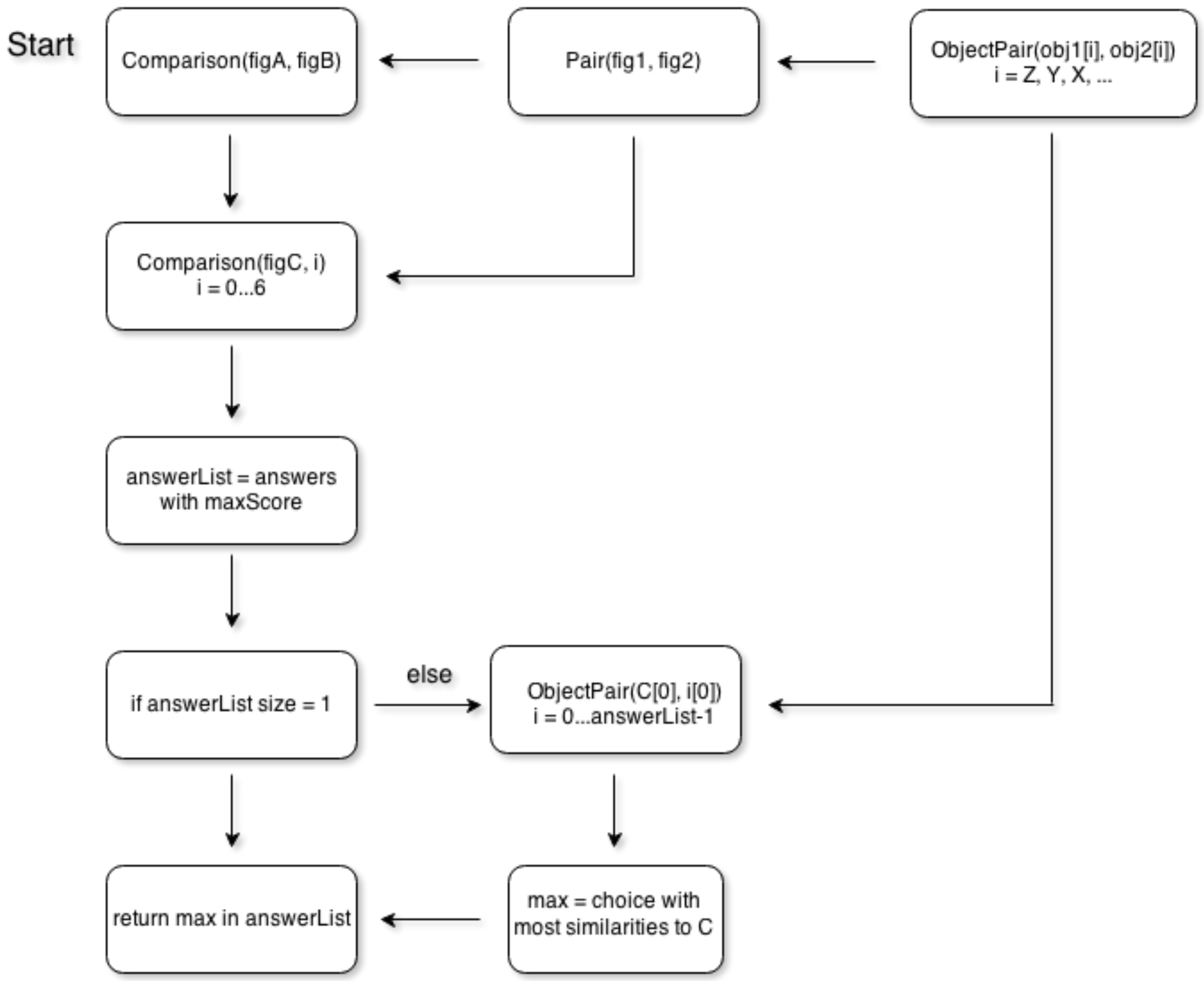
The purpose of this class was to track the relationship differences between each object in a figure. This class kept a hash map of characteristics and an associated boolean value determining if they were the same or different. The key values in the hash map were strings of the 4 key characteristics identified: shape, size, fill, and angle.

Variables

obj1: the first object to be compared
obj2: the second object to be compared
obj1Attributes: a hash map containing all of the attributes of obj1
obj2Attributes: a hash map containing all of the attributes of obj2
sameShape: a boolean value determining if each object contained the same shape
sameFill: a boolean value determining if each object contained the same fill
sameSize: a boolean value determining if each object contained the same size
sameAngle: a boolean value determining if each object contained the same angle
objValues: hash map containing each of the key attributes and a boolean value associated determining if they were equal

Key Functions

calcSameShape: determined if the object contained the same shape and stored that boolean value in the hash map.
calcSameSize: determined if the object contained the same size and stored that boolean value in the hash map.
calcSameFill: determined if the object contained the same fill and stored that boolean value in the hash map.
calcSameAngle: determined if the object contained the same angle and stored that boolean value in the hash map.
getSame… : getters for each of the boolean attributes comparisons
getValues: getter for the hash map containing all of the attributes and their boolean values

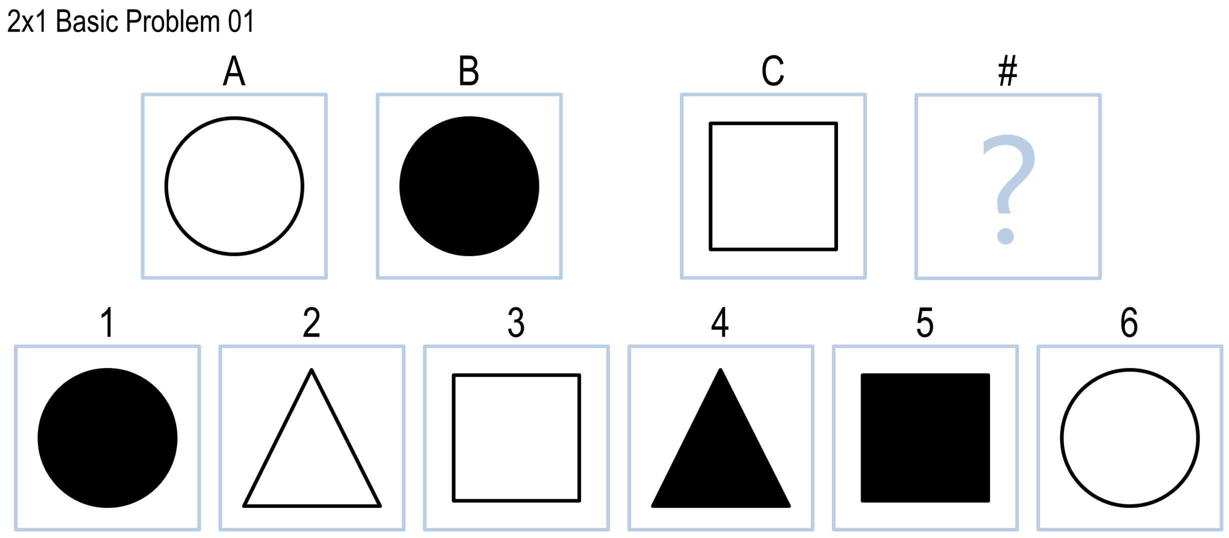


Flow Diagram of how Solve finds Answer

2. Application

We will walk through an example using the strategy I implemented to show how the agent solves a Raven's progressive matrices problem.

A. Example Problem Walk-Through



Comparison(figA, figB)

Because this A and B only have one shape, this process will be much easier. figA will compare to figB it's shape, size, fill, and angle, and then store these in a hash table. The table should look something like this:

Hash Map of Figures A and B	
Shape	TRUE
Size	TRUE
Fill	FALSE
Angle	TRUE

Comparison(figC, i)
i = 0…6

Therefore, when we run the comparison on all of the answer choices to C, we will find that there are 3 answers that yield 3 True values and 1 False. The 3 figures that will be added to answer list are 2, 5, and 6. The reason 3 will not be added is because it has 4 True values, which is one more than we are looking for.

$$\begin{tikzcd}
\text{If answer list size = 1} \arrow[r, "\text{else}"] & \text{ObjectPair(C[0], i[0]) for i=0…answerList-1}
\end{tikzcd}$$

Because our answer list will contain 3 values, we will iterate through the first object of C and compare the similarities it has with the first objects of figures 2, 5, and 6. This first object is named Z in each of the figures, based on the API we have been given. After running these calculations, we will find that:

Scores of Figure 2, 5, and 6	
Figure 2	0
Figure 5	3
Figure 6	0

The reason Figures 2 and 6 have a score of 0 is because they don't contain the same shape as Figure C, which is a heuristic I implemented. Because A and B have the same shape type and number of shapes, I disqualified any candidates that didn't at least replicate those standards. This leaves only Figure 5 as a candidate for the answer.

return answer = "5"

3. Shortcomings

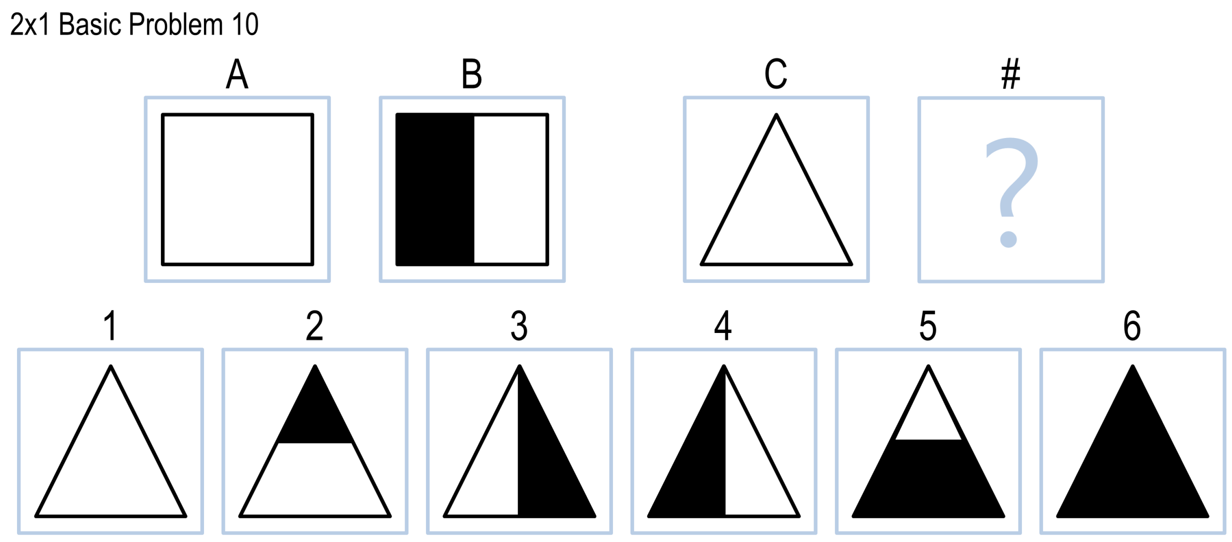
Obviously, this agent does have some downfalls. One of the biggest issues is determining not just that a change occurs, but where it occurs and also to what extent. My agent knows that 3 out of 4 things may change from A to B, and even knows the categories of these changes, but doesn't know what changed.

A. When does it fall short?

My agent usually fails when there are multiples object involved in the change, and when the position on the screen becomes a factor. My agent can detect when there are 3 objets each of triangle, square, and circle, but doesn't know their position on the screen in relation to each other. That makes it difficult with some of the more challenging problems, where you not only have to know what changed about each shape, but to the extent of the change. It also doesn't have a sophisticated numerical system for rating shapes. Right now, my agent just stores boolean values in major categories highlighting if a change occurred. It doesn't comprehend the extent of the change, making it hard to choose the right figure when many of the answer choices are similar in the ways they are different from C.

B. Why?

I thought it would be easiest to demonstrate this through a short example.



Right now, my agent will store figures 2, 3, 4, 5, and 6 as possible answers in my answer list. The reason being that it calculates from figure A to B that only the fill changed, so it excludes figure 1 for being a possible answer. However, it has no idea now how to determine that only the left-half of the shape was filled, making it extremely easy for the human eye to choose figure 4 as the right answer. My agent will return figure 2 because it stores boolean values saying the fill is not the same, but it doesn't know how to distinguish in what way they are different. Therefore, it will view them all the same, and because 2 appears first in the list, will return that as the answer.

4. Potential for improvements

A. What could have been done better?

There are so many ways in which this agent could have been improved. The easiest way is to highlight it's scoring system for potential answers. One of the biggest shortcomings in my agent is that it just stores a boolean value for the difference between figures in major categories of objects. One way to improve on this would be to implement a scoring system for the amount of change between major categories in attributes of an object. Obviously this would have taken extra classes, coordination, and planning, but would have been the most effective way in determining a heuristic from one figure to another.

So instead of passing in boolean values with the differences between the "shapes" of two objects, the scoring system could have perfectly assigned a numerical distance. This would have made the eliminations of certain answers must easier to weed out.

Finally, instead of making one pass through the answer choices and eliminating them based on attributes calculated at once, we could have made many passes, eliminating figures by one characteristic. For example, the first pass would have eliminated each figure that didn't contain 3 objects, then check the remaining to see if they all had square objects, and etc. This would have been easy to try and narrow down an answer by process of elimination, just continually looping until you have one answer left. Of course, in order to know what to eliminate, you would have to understand the relationship between figures A and B, and then understand figure C as well.