Eugen Istoc
CS7637

Project 1

Introduction

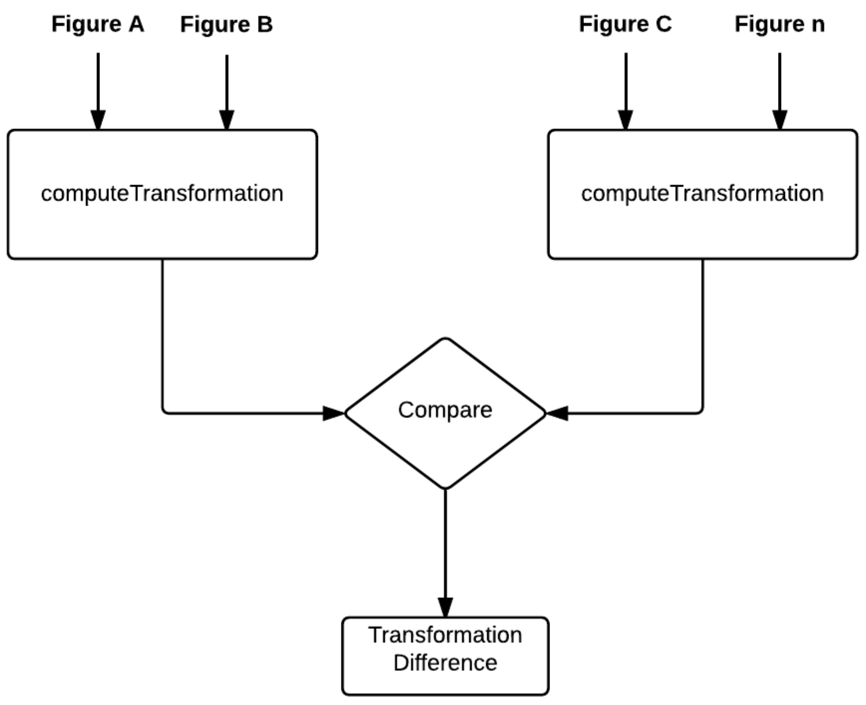
The approach that I took in designing and building this agent in solving *Raven's Progressive Matrix problems*, or *Visual Analogy problems*, is to translate the theoretical ideas of problem generation/testing and means end analysis into code. This process is a significant challenge due to the fact that many times when speaking of these ideas at a high level abstracts out many of the limitations that are more often than not evident only when one begins to implement said solution. For example, in my design, I leverage the *ArrayList* object very often in maintaining lists of information that my *Agent* keeps track of. I had to introduce, however, extra boilerplate code when working with said data structure in order to enable the agent to function according to the rules of the problem.

Assumptions

In approaching a particular problem set, there are a few assumptions that the agent makes. For example, it is assumed that figure **A** and **B** are always the 2 input (given) figures. Furthermore, figure **C** is always the figure from which a valid transformation must be applied to it to arrive a possible answer. There is no assumption on the number of possible answers. Even though, in the problem sets given there are always 6 answers, the agent treats all the figures, with the exception of figure **A**, **B**, and **C** as possible answers.

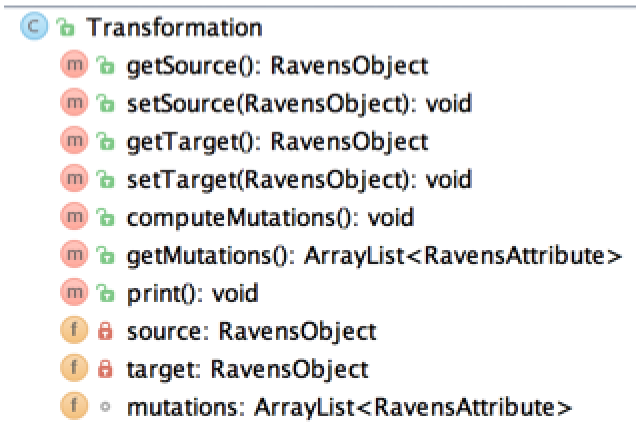
Agent Architecture

In general, the Agent has two major components. The first component calculates a transformation between two figures, and the second component tests two transformations. This model of generate and test is common, and is illustrated below as it relates to this agent:



In essence, a transformation is calculated between **Figure C** and all of the possible figures **1..n**. Each transformation, is then compared to the *sourceTransformation*, that is, the transformation between **Figure A** and **Figure B**. The transformation which yields the least amount of difference to the *sourceTransformation*, is considered to be the best answer[1]. When computing transformations, the agent implements a somewhat intelligent generator, as it considers non-uniform object mappings. This is crucial when the agent builds a list of *candidateAnswers*, as it reduces the margin of error in selecting an answer (See *Reasoning* section).

When speaking of a transformation, the Agent utilizes a *Transformation* data structure which intelligently maps a *source* object to a *target* object, and calculates the mutation between them. This data structure is illustrated in the figure to the left. Note that a mutation refers to a change in attributes between the *source*, and the *target*. I use the term *mutation* to imply that the *source* and the *target* map to each other 1-1, and all the attributes that exist on the *target* and NOT in the *source*, are changes.

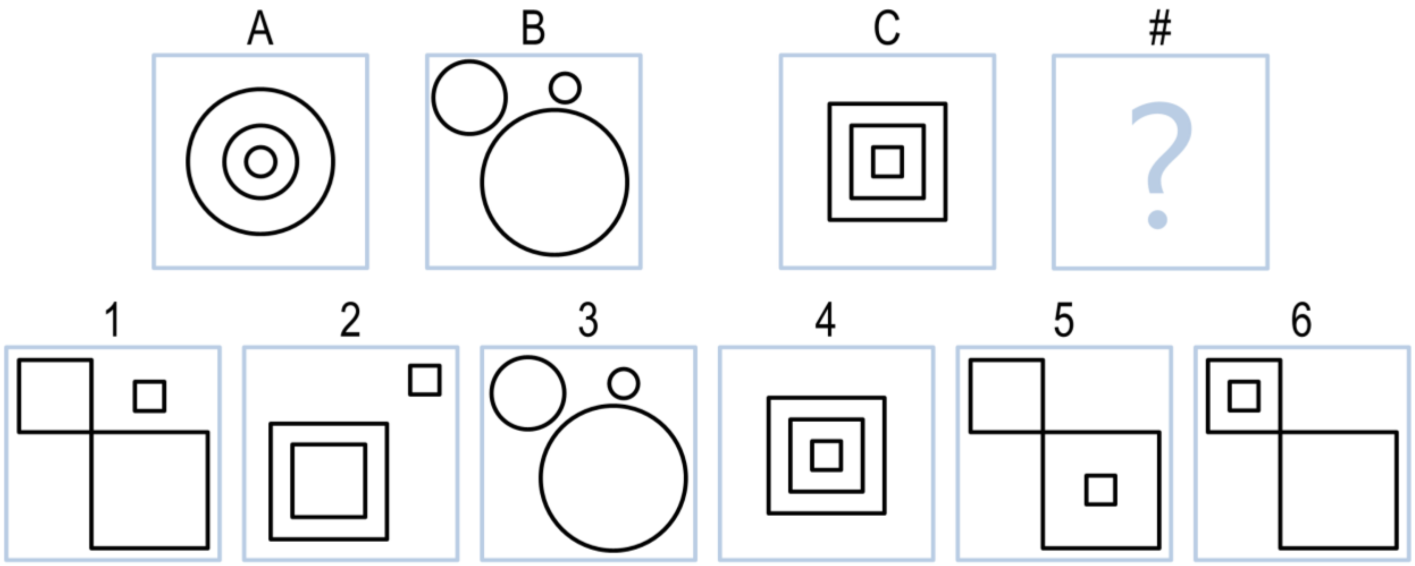


```
Transformation
    getSource(): RavensObject
    setSource(RavensObject): void
    getTarget(): RavensObject
    setTarget(RavensObject): void
    computeMutations(): void
    getMutations(): ArrayList<RavensAttribute>
    print(): void
    source: RavensObject
    target: RavensObject
    mutations: ArrayList<RavensAttribute>
```

Reasoning

An important aspect of generating answers is to consider edge cases. For example. some problems involve the removal of shapes from one figure to another. The agent must figure out how to map, for example, 2 source objects to 1 target object. This is accomplished during the *computeTransformation* stage. When analyzing each obejct in a target figure, the agent assigns weights to every object so that it can determine which of the target object it maps to. The weights are determined by the similarities in the attributes between the *source* object and the *target* object. For example, consider problem *2x1BasicProblem08* below:

2x1 Basic Problem 08



To determine which object in **A** maps to which object in **B**, for example, the agent would assign the following weights:

[1] There are cases when there are multiple possible answers. This happens when multiple transformations yield the same difference (Rather than a single transformation difference that happens to be the lowest).

$$\begin{tikzcd}
 & Z\ 0 & & Z\ 2 & & Z\ 2 \\
Z \arrow[ur] \arrow[r] \arrow[dr] & Y\ 1 & Y \arrow[ur] \arrow[r] \arrow[dr] & Y\ 1 & X \arrow[ur] \arrow[r] \arrow[dr] & Y\ 2 \\
 & X\ 1 & & X\ 2 & & X\ 1
\end{tikzcd}$$

The weights outlined in green indicate which object in **Figure A** maps to which object in **Figure B**. In this case, we can see that objects seem to map 1-1. How does the Agent know how to assign these weights to the objects? This is accomplished by comparing the attributes in the source object, to the attributes in the target object. In the table below, I illustrate how these attributes are compared.

Figure A	Object	Attributes	Figure B	Object	Attributes	Differences

	Z	shape(circle)
		size(large)
				Z	shape(circle)
					size(large)
						0

	Y	shape(circle)
		size(medium)
		inside(Z)
				Y	shape(circle)
					size(medium)
					above(Z)
					leftOf(X)
						1

	X	shape(circle)
		size(small)
		inside(Z,Y)
				X	shape(circle)
					size(small)
					above(Z)
						1

As shown above, only attributes found in **Figure A**, but not found in **Figure B** are considered as differences.

Correct Answers

My agent is able to solve, according to the problem set given about 75% of the problems. There are a few limitations in which the agent fails to arrive at the correct answer which are described below, but in general the agent's approach is viable, and given more time, these limitations can be mitigated. The way that the agent is designed, enables the system to be further extended by recursively solving a problem set (See *Improvements and Optimizations below),* however, keeping that in mind, some of the correct answers that the Agent produces are simply guesses. A guess is made when there are multiple *candidateAnswers*. Note, however, that although this is just a guess, it is somewhat educated. On most of the problems that involve guessing the correct answers, that Agent would have already removed obvious wrong answers from the *candidateAnswers* set. Therefore, the likeliness of a correct answer is certainly higher.

Mistakes

The main source of mistakes for this Agent is the fact that attribute values are hardly considered in most of the computations. For example, each object properties is composed of an attribute of the format "**attributeName: value**". The *value* attribute is something that the agent treats very lightly. This causes it to make mistakes on problems where attribute values are crucial to the answer. An example of a problem like this is *2x1BasicProblem12*. In this problem, the transformation seem to only be between the various shapes. There are hardly any translations or scaling. This problem, therefore, corners the agent and it produces an incorrect answer. This is the only major mistake that the agent seems to make, and it's certainly one that can be remedied given more time to tweak the algorithms in question.

Improvements and Optimizations

As stated earlier, when a problem is not immediately solvable, the Agent creates a list of possible answers (*candidateAnswers*). When this happens, the Agent simply makes a guess at the answer with the hope, obviously, of guessing the right answer. A very obvious optimization, at this point, is to feed this answer back as an input to the agent for further analysis — until the number of candidate answers is reduced to 1 (the correct answer. This optimization can be implemented recursively for the kind of problem sets which are inputted.