Fardad Hajirostami

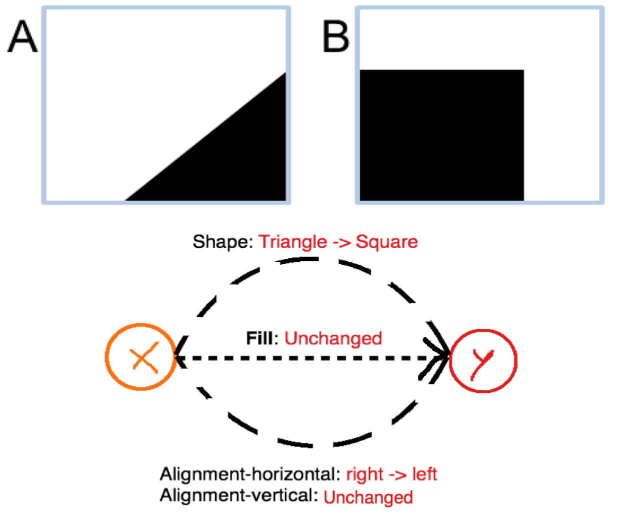
Project 1 Reflection

I. Introduction

The goal of this paper is to provide a holistic report of the approaches, designs, methodologies, and challenges that underwent in making an agent capable of successfully tackling and solving 2x2 Raven's Progressive Matrices (RPM). I will first provide a concise summary of our given problem and the task that is at hand. We will look into the thought process that took place in strategizing an appropriate implementation for our agent. Next, we will provide a detailed explanation of the backbone of our agent's implementation (i.e. how our agent is going to receive and store information) followed with their justifications. Later, we will explore our agent's high-level design and the multitude of strategies and paradigms it uses, some of which we have discussed thus far in KBAI in addition to supplementary tactics to ultimately lead our agent to arrive at the correct (or at least close to correct) answer. Finally, we will analyze the performance and accuracy of our agent and close our discussion by reflecting back on the challenges and limitations that our agent faced as well as how we will address these conundrums moving forward.

II. Problem Description and Planning

We are given a training set of twelve 2x2 RPM's and information regarding each RPM is presented to us both textually and visually. The textual description of our RPM's include information regarding at least 3 features (i.e. shape, size, fill…) of each Raven object in each Raven figure. The very first task that is given to us is deciding if our agent will have sufficient knowledge regarding a given problem if it's only presented with verbal knowledge. We can do this by going through each problem and mentally visualizing how our agent will make sense of its world. How will it depict the transformations that take place between each 2x1 problem (A -> B or A -> C) and how will these transformations will map out from B -> D or C -> D where D is our proposed solution? To answer these questions, we shall begin by utilizing fundamental concepts discussed in KBAI via formulating a mental construction of semantic network for sample Raven's problems. Additionally, we must detect whether or not the textual representations alone can suffice. To demonstrate this process, let's look at a specific example:


A
B
Shape: Triangle -> Square
Fill: Unchanged
Alignment-horizontal: right -> left
Alignment-vertical: Unchanged

Verbal Representation:

A
- **a**
  - **shape:right triangle**
  - **size:very large**
  - **fill:yes**
  - **angle:270**
  - **alignment:bottom-right**

B
- **b**
  - **shape:square**
  - **size:very large**
  - **fill:yes**
  - **alignment:bottom-left**

Figure 1

We can apply this paradigm of semantic network construction to other examples in our training set and come to the safe conclusion that verbal representations alone sufficiently generalize and represent both our training and test sets. Of course, we must also take into consideration the fact that certain verbal descriptions must be interpreted into a vocabulary that will make sense for our agent. Let's look at a slightly more complicated example:

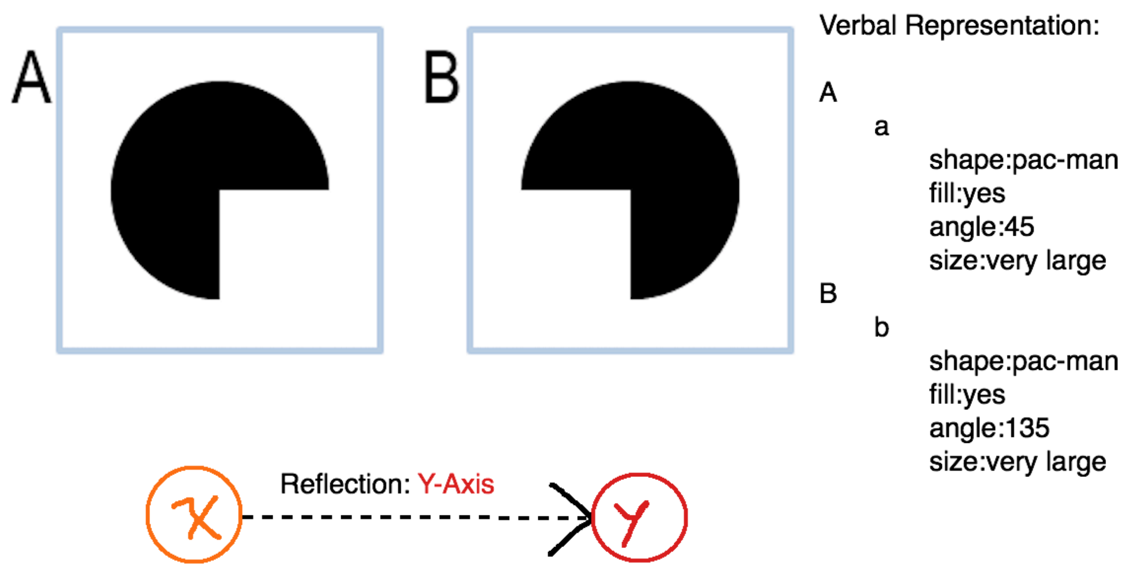
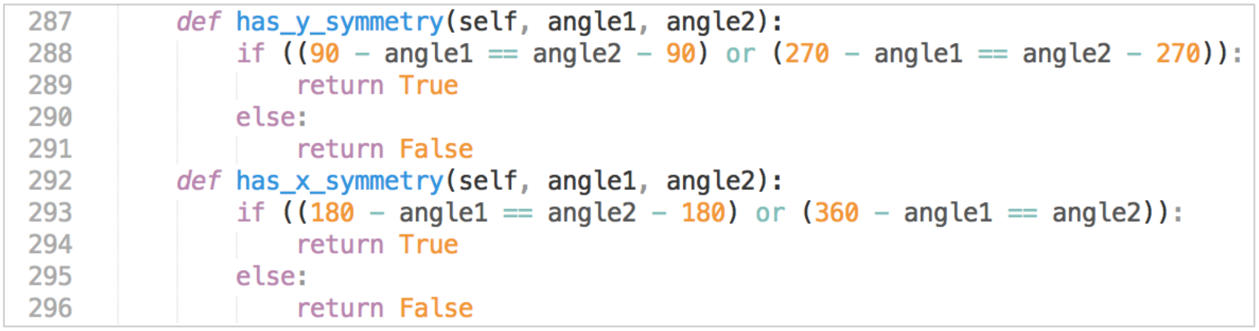


Figure 2

Upon first glance, this looks like a simple problem and we might naïvely come to the conclusion that reflection across Y-axis can be simply represented as the following: new_angle = original_angle + 90. However, this formula will not take into account for all cases of reflection across the y-axis. If we flip this problem by replacing A with B and vice versa, then we can clearly see that while both images are still horizontally reflected, our formula will lead us to the wrong answer. Therefore, to correctly translate horizontal and vertical reflection based on an object's angle, we arrive at the following lines of code that consider all possibilities for both horizontal and vertical reflections:



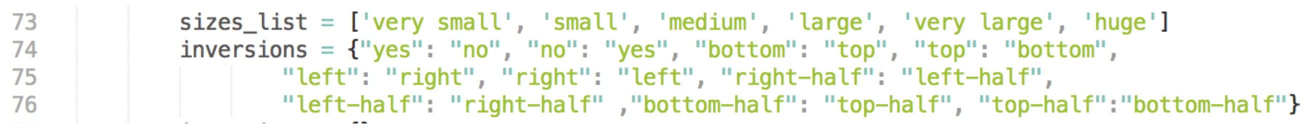
```
def has_y_symmetry(self, angle1, angle2):
    if ((90 - angle1 == angle2 - 90) or (270 - angle1 == angle2 - 270)):
        return True
    else:
        return False
def has_x_symmetry(self, angle1, angle2):
    if ((180 - angle1 == angle2 - 180) or (360 - angle1 == angle2)):
        return True
    else:
        return False
```

Figure 3

By applying these abstractions and translations, we are one step closer to designing an agent that effectively generalizes the transformations that take place in our semantic networks. Next, we will take a deeper look into how our agent stores and organizes information in addition to a step by step outline of how it generates and tests solutions.

III. Agent Architecture and Reasoning

Since we are dealing with lots of complex and different transformations, it can be easy for our code to become messy and thus it is important to keep code redundancy to the minimum. We also want to conserve memory and avoid recalculating or retrieving information that we have previously obtained in the past. One way of doing this, is by creating a dictionary/list of relevant transformation data that our agent might need in the future:



```
sizes_list = ['very small', 'small', 'medium', 'large', 'very large', 'huge']
inversions = {"yes": "no", "no": "yes", "bottom": "top", "top": "bottom",
        "left": "right", "right": "left", "right-half": "left-half",
        "left-half": "right-half" ,"bottom-half": "top-half", "top-half":"bottom-half"}
```

Figure 3

For example, we have included the descriptions of sizes from ordered from smallest to biggest to make it easier for us to make future size comparisons based on their index. In order to map out the transformations of A -> B or A -> C, we must first generate a “transformation dictionary” that will store features descriptions such as “shape”, “size”, “angle”, and etc. in its key set and pair them with their appropriate transformations value:

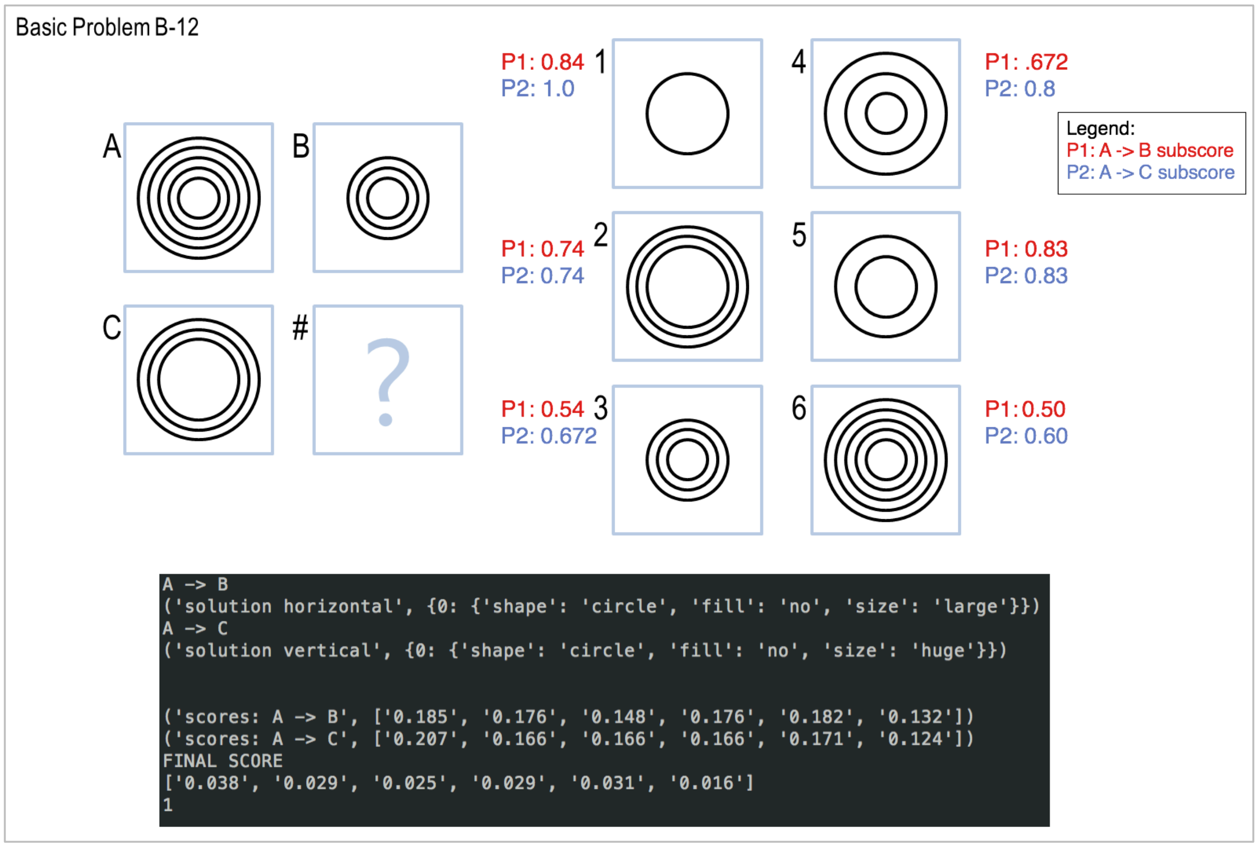
“size” -> {-5, 5} (# range representing sizeDelta)
“alignment” -> {“V”, “H”, “VH”}

One particular challenge in designing our agent was in representing deletion or insertion transformations. Initially, we can easily depict whether or not a deletion or insertion takes place by in a given 2x1 problem by simply comparing the length of our figures. Next comes the more challenging task of deciding exactly which object got deleted or inserted. We must keep in mind that every Raven figure has its own unique raven objects with unique keys. While it might be tempting to make the assumption, we cannot guarantee whether or not there’s any relation between the lexicographical ordering of the keys of different Raven’s objects. To tackle this puzzle, each Raven object in figure A must be compared with every object in the figure that we are comparing to in order to ultimately capture their similarity scores. To calculate similarity scores, we use probabilistic reasoning by weighing different features based on their influence on their effect on determining whether or not an object is deleted or inserted. The following table depicts the ideal weights that were assigned to each feature transformation based on both instinctive reasoning and testing with varying weights:

Weight	Transformation
0.8	Shape-Change
(.8 - (0.04*d))	Size-Delta (d)
0.6	Fill-Change
(0.6 - (0.04*d))	Inside-delta (d)
0.5	Rotation

Table 1

The different weights essentially tell our agent how heavily it should “punish” or “reward” each Raven’s object chance of later being selected or deselected as a potential candidate for our deleted or inserted object. After we finalize our transformations dictionary, the next task that our agent must perform is to generate a potential candidate solution by appropriately mapping out the transformation matrix on the third figure (aka the figure that must undergo a similar transformation to A). After a solution dictionary has been generated, every object that is represented in our solution dictionary will be tested with every object in each solution figure. For every test out of the six possible tests, we calculate a probability sub-score for each object in our solution dictionary. This sub-score is calculated by taking the maximum probability that our object was able to achieve after being compared to its closest match. After we calculate the sub-scores of each Raven Object in our solution dictionary, the sub-scores of all the raven objects will be multiplied together to give us the final probability of our generated solution for the particular solution candidate.

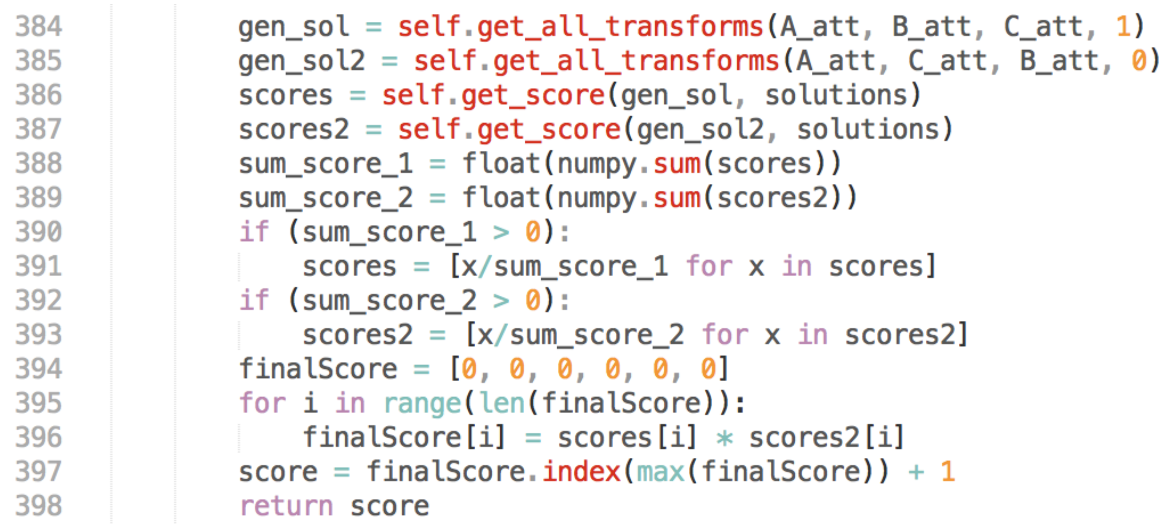


```
A -> B
('solution horizontal', {0: {'shape': 'circle', 'fill': 'no', 'size': 'large'}})
A -> C
('solution vertical', {0: {'shape': 'circle', 'fill': 'no', 'size': 'huge'}})


('scores: A -> B', ['0.185', '0.176', '0.148', '0.176', '0.182', '0.132'])
('scores: A -> C', ['0.207', '0.166', '0.166', '0.166', '0.171', '0.124'])
FINAL SCORE
['0.038', '0.029', '0.025', '0.029', '0.031', '0.016']
1
```

Figure 4

It should also be noted that an additional weight was added to our probabilistic calculation by computing the absolute difference between the length of our generated and candidate solution and subtracting this weight by our sub-score prior to appending it to our finalized sub-score list. The reasoning behind this logic is to punish our agent whenever it finds disparities in the total number of objects both our generated and test solution hold. Without doing this, our P1 and P2 scores for solution 6 shown in *figure 5* will end up being undesirably high since our agent will record the highest probability of the closest match of one of the objects in the Raven figure while disregarding the rest. Once the sub-scores for each potential candidate have been finalized for both P1 and P2, the scores for both A -> B and A -> C shall be normalized and multiplied together to yield a finalized list whose maximum indicates the solution to our RPM! The following snippet includes the final steps our agent undergoes to arrive at the final answer.



```
gen_sol = self.get_all_transforms(A_att, B_att, C_att, 1)
gen_sol2 = self.get_all_transforms(A_att, C_att, B_att, 0)
scores = self.get_score(gen_sol, solutions)
scores2 = self.get_score(gen_sol2, solutions)
sum_score_1 = float(numpy.sum(scores))
sum_score_2 = float(numpy.sum(scores2))
if (sum_score_1 > 0):
    scores = [x/sum_score_1 for x in scores]
if (sum_score_2 > 0):
    scores2 = [x/sum_score_2 for x in scores2]
finalScore = [0, 0, 0, 0, 0, 0]
for i in range(len(finalScore)):
    finalScore[i] = scores[i] * scores2[i]
score = finalScore.index(max(finalScore)) + 1
return score
```

Figure 5

IV. Reflection

In the end, our agent was generally successful in arriving at the correct solution. For the given training set, our agent was successful 11 out of the 12 times. The agent had difficulty with Basic Problem 10 as it included a combination of insertion and deletion transformations and thus it had difficulty coming up with an answer that included the union of both transformations. Additionally, our agent's performance declined when presented with the test set as it was only able to solve 8 out of the 12 given problems. One obvious reason behind the decline in our agent's performance could be due to the fact that our implementation over fitted the training examples due to being too specific in our semantic construction. This is a common challenge that we often face in KBAI as it is often hard to draw the line between when we are being too specific vs when we are over generalizing. The ultimate goal of knowledge representation via semantic networks is to "provide a level of abstraction at which the problem gets represented and analyzed" (P. 49, Goel, Ashok; Joyner, David, 2017). When thinking back, one possible wrong doing in our semantic network construction could have resided in our shape change transformations. Specifically, whenever there is a shape change from A -> B or A -> C, we recorded this transformation precisely as shape1-to-shape2. As a result, our agent potentially was too “nit-picky” in that it carried out the mapping this transformation if and only if it encountered shape1 again. What if the true shape transformation for a given RPM was square -> triangle and pentagon -> square? (aka changes in number of edges). Certainly, with our agent’s current implementation, it will undoubtedly glace over such an abstraction. Other possible theories for our agent’s shortcomings could be due to poor knowledge representation that is depicted by our transformation weights. Perhaps, we over fitted our training set once again but this time due to the fact that our proposed weight scheme worked too perfectly for our training set while lacking abstraction of the influence of each transformation to a wide array of RPM problems. What if instead of keeping our weights constant, we try dynamically changing them according to the feedback it receives on its performance on each subsequent problem? Perhaps this could be achieved by constructing a neural net in which Raven object features indicate our input nodes and finalized transformations depict our output nodes. While neural nets can yield highly accurate results, we must also take into consideration the speed at which our agent is able to come up with a feasible solution. Construction of accurate neural nets can be notoriously long especially as the total number of our nodes and hidden layers’ increase. An even better future step might be implementing a Q-learning algorithm and having our agent learn from its failures in a state by state matter. In his “Bridging the Gap between Reinforcement Learning and Knowledge Representation”, Emad Saad demonstrates his success in building a model-free reinforcement learning agent capable of successfully solving encoded SAT problems. Certainly, by providing our agent with an appropriate reinforcement learning paradigm, one which punishes and rewards our agent rationally, we should be able to solve even the most complex RPM problems.

References:

KBAI Ebook: Knowledge-based Artificial Intelligence ©2016 Ashok Goel and David Joyner

Saad E. (2011) Bridging the Gap between Reinforcement Learning and Knowledge Representation: A Logical Off- and On-Policy Framework. In: Liu W. (eds) Symbolic and Quantitative Approaches to Reasoning with Uncertainty. ECSQARU 2011. Lecture Notes in Computer Science, vol 6717. Springer, Berlin, Heidelberg

Findler, Nicholas V., ed. (1979) Associative Networks: Representation and Use of Knowledge by Computers, New York: Academic Press.