Project 2 reflections

**I. Introduction**

Raven's progressive Matrices (RPMs), written in 1930s, are multiple-choice questions composed of non-verbal presentations that ask the subject to identify the missing element according to their patterns. They are generally in forms of 2x2, 3x3 or even more complex matrices. RPMs are "the gold standard in intelligence test" -- according to CNN. Thus they have been used to test IQ value of children, elderly and possibly mentally-impaired individuals. One popular aspect of AI is knowledge-based AI is based on unifying Reason-Learning-Memory processes, mimicking how human brain processes knowledge from the environment. Since the agent we are building will exhibit human-like intelligence, RPMs are good tests to examine and compare how good our strategies are in terms of making the agent "intelligent".

There are many difficulties in solving the RPM problem. For example, how can we represent the problem in a way that computer can easily understand? How to determine the corresponding relations between different objects in these panels? How to come up with transforms and examine their similarities? How to break ties when similar options come up? In project 2, starting with the simpler model implemented in project 1, based on the new requirements and problems encountered, I improve my AI agent to solve the 2x2 as well as those more complicated 3x3 RPM problems.

**II. Data structure**

First, a recap of the data structure used in this project:

Overall, the problems are represented by "Frames" or classes in OOP, and The information about the RPM problems is stored in a hierarchy of classes:

Main 🡪 ProblemSet 🡪 Individual Raven Problem 🡪 Raven Figure 🡪 Raven Object 🡪 Attribute

* The agent starts from the **main** method in RavensProject.java and reads in all the RPM problems.
* These problems are then stored in ProblemSet.java. The **ProblemSet** is composed of a ProblemSet name (such as “Basic Problem C” or “Challenge Problem C”) and a list of Raven problems.
* Individual Raven Problem is represented by **RavensProblem**.java, which consists of problem name, problem type (such as “2x2” or “3x3”) and a HashMap storing different panels of figures.
* Each **RavenFigure** is composed of a figure name (such as “A”-“H” or “1” or “8”), and a HashMap of Raven objects.
* A **RavenObject** is represented by RavensObject.java, which contains object name (such as “a”, “b”), and a HashMap storing all the **attributes** of this object (such as “shape: circle”).
* Finally, to record transformations, I constructed a new class named **Transformation.java**. This class contains two RavenObjects, a string list of mutations observed in this particular transformation.

**III. Core algorithms**

**1. Method implemented**

In my agent, I implemented in method called computeTransformation() to compute the transformation between figures:

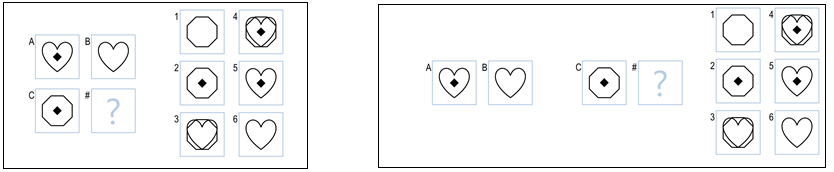
**private** ArrayList<Transformation> computeTransformation(RavensFigure A, RavensFigure B)

and another method to score differences between transformations:

**private** **int** computeDifference(ArrayList<Transformation> targetTransforms, ArrayList<Transformation> choiceTransforms)

**2. Converting 2x2 to 2x1**

For a Raven 2x2 problem shown in **Figure 1**, I first call computeTransformation() to compute the transformation between Figure A and B, as well as transformations between Figure C and all the candidate answers. Next, I call computeDifference() to evaluate how similar these 6 candidate transformations are compared to the expected transformation from A to B. Finally, I iterate to pick up the candidate answer with the lowest score and set it to be the given answer of the problem.



**Figure 1 Figure 2**

Apparently, the 2x2 RPM problem was considered as a 2x1 problem this far (**Figure 2**). To solve it as a 2x2 problem, I repeated previous procedures by computing the transformation between A and C, and also the transformations between B and the candidates. Similarly, I rate how similar all 6 transformation (from Figure B to a candidate answer) are compared to transformation A to C. In the end, I iterate two scoring list (A🡪B and A🡪C) and pick up the best answer and set it to be the given answer of the problem.

**3. Why are 3x3 problems harder?**

The 3x3 Raven problems are generally harder than 2x2, the difficulties come from:

**1) More transformations need be generated:**

The transformation between A and B tends to be ambiguous and arbitrary in 3x3 problems, and one needs to further examine the changes between B and C to narrow down the scope of possible transformations. This requires our agent to have a larger pool of transformation candidates to choose from, so that the agent could have a higher chance to end up with a transformation that fits better.

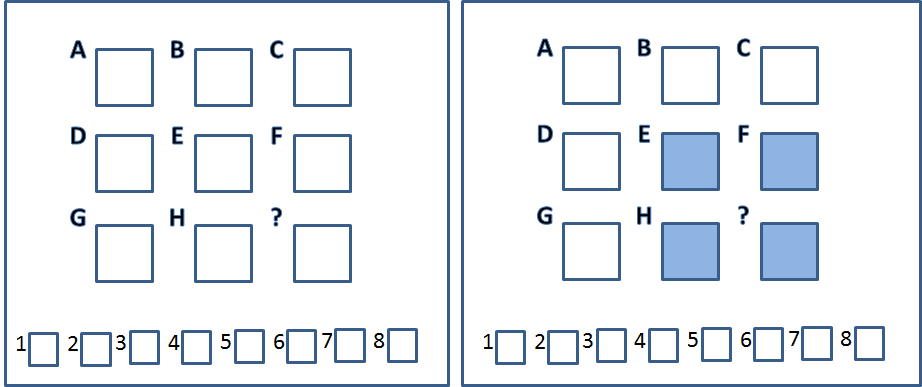
**2) More transformations to evaluation:**

Even if we start with a larger transformation pool, we still need to come up with more sophisticated strategies about how to rank all the possible transformations based on information from different rows and different columns, how to break ties, and etc.

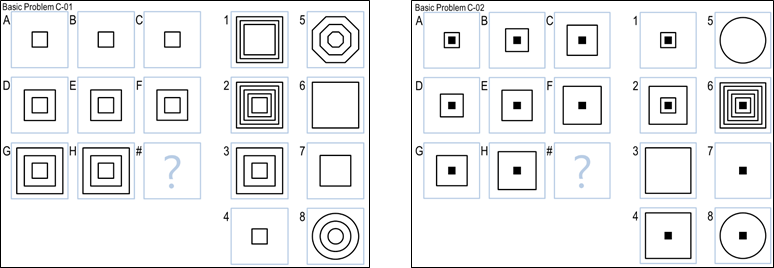
**3) More complicated networks between panels:**

Besides the horizontal and vertical relationship, we also need to take the diagonal transformation into consideration.

**4. Converting 3x3 to 2x2**

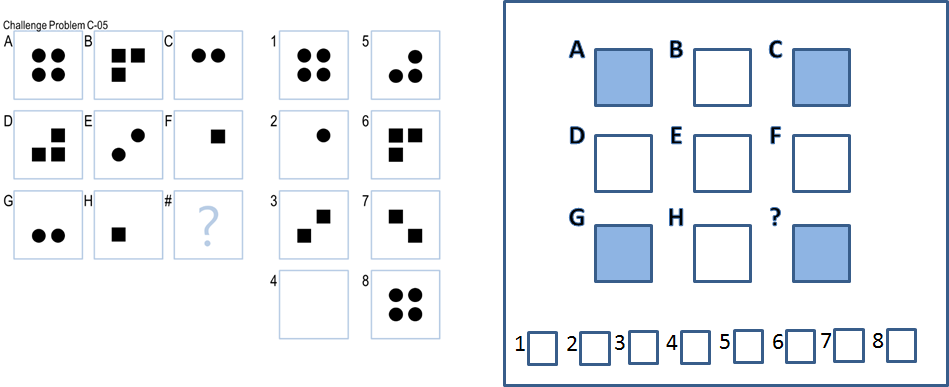
****

**Figure 3**

Inspired by how we converted 2x2 problems into 2x1 problems, I decided to try to solve the 3x3 RPM problems by simplifying them into a few 2x2 problems. For example, I can ignore some of the figures in the 3x3 problem shown in **Figure 3**-left panel, and only consider it as 2x2 problem with 8 choices as shown in **Figure 3**-right panel. This strategy can solve some simple 3x3 problems as shown in **Figure 4 and 5** (Basic Problem C-01 and C02). In these problems, the transformations between E🡪F (corresponding to A🡪B for 2x2) is either “no change” or “enlarge”, and the transformations between E🡪H (corresponding to A🡪C for 2x2) is either “appear” or “enlarge”. These transformations extracted from the E-F-H-# 2x2 sub-problems have already summarized the general transformation trends in the whole 3x3 problems, and at the same time remain specific enough for the agent to pick the best matching answer. ****

**Figure 4 Figure 5**

The lower-right corner is not necessary the best representation of a 3x3 problem. Sometimes, a 3x3 problems can be better solved by converting to a 2x2 problem with another subset of panels. For example, in the following example shown in **Figure 6**, if we look at the E-F-H-# sub-problem, it is hard to define the appropriate transformations that represent the whole problem. However, if we look at the A-C-G-# subset, then the transformation is simply “disappearing of two circles” both horizontally and vertically. Thus #4 is chosen as the best answer.

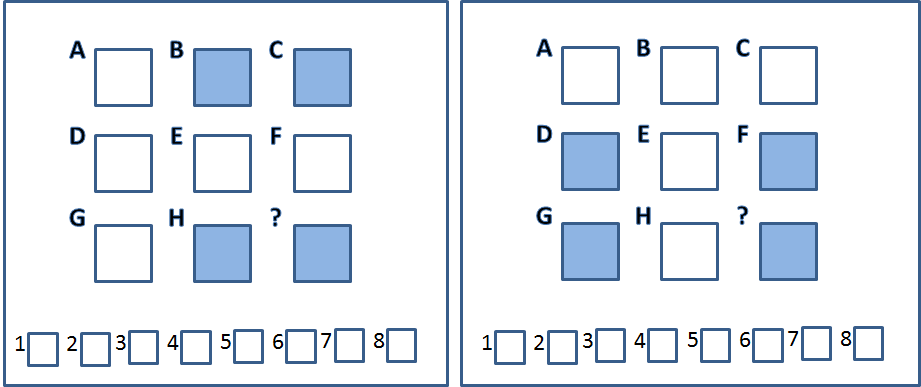
****

**Figure 6**

Similarly, a 3x3 problem can also be simplified as

1) horizontal and vertical

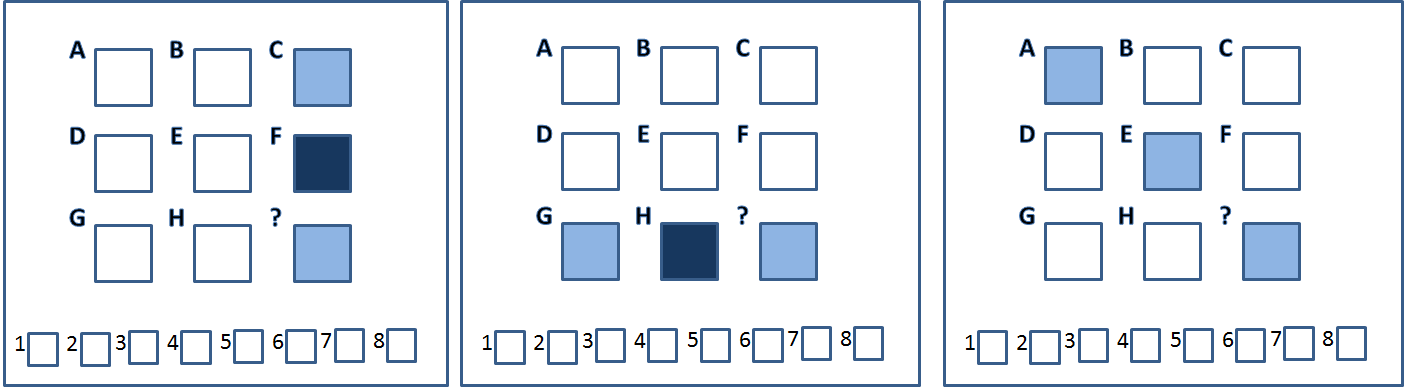
**B-C-H-# or D-F-G-#**



**Figure 7**

2) one row/column/diagonal

**C-F-F-# or G-H-H-# or A-E-E-#**



**Figure 8**

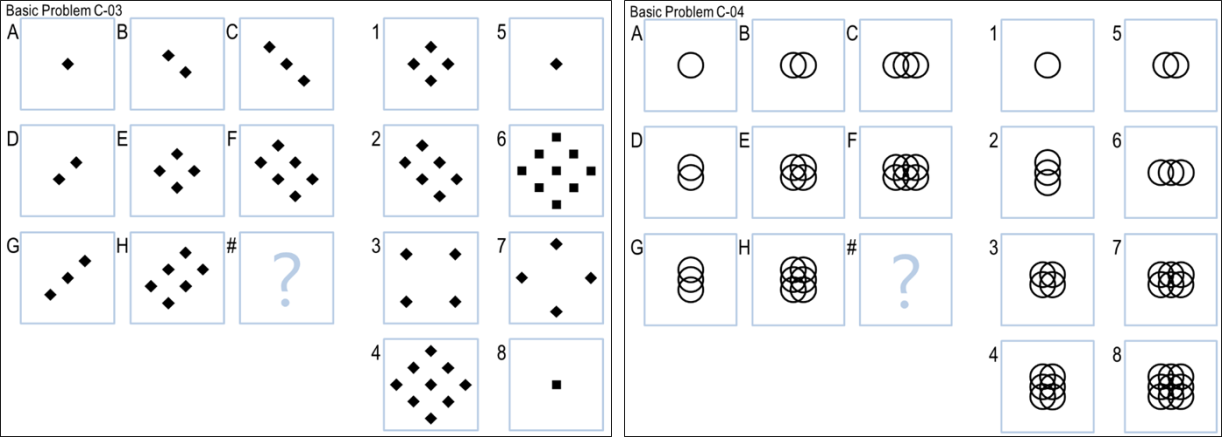
**5. How to combine the answers?**

In each of the subset generated by the original 3x3 problem, the agent will pick up the best answer in that scenario. Since every 3x3 problem is different, generally we don’t know which subset will be better than others before a specific problem is encountered. So the next question is how the agent should combine these results? One straightforward method is to average the scores of each choice. Another way is to iterate through all scenarios and to pick up the one with the lowest score as answer of the problem. These two methods both have advantages and drawbacks. I chose to use the later one when implementing my agent.

**IV. Improvements during implementation**

**1) “single-object” problems**

One of the biggest obstacles in project 2 is the performance of my agent is particularly bad about a specific group of problems, which I named them **“single-object” problems**. An example can be seen in **Figure 9** (Basic Problem C-03 and 04). In this kind of problems, the figures are composed of the same object (specifically defined as objects with the same shape, size and filled status), and the only difference is the number of objects in each panel. The reason my agent is bad about these problem is : the foundation of my AI agent is largely based on correct matching of objects between different panels. If objects appear/disappear in a transformation, the penalty score is pretty high. So my agent fails to solve “single-object” problems, due to the noise raised by multiple “appear/disappear” mutations.

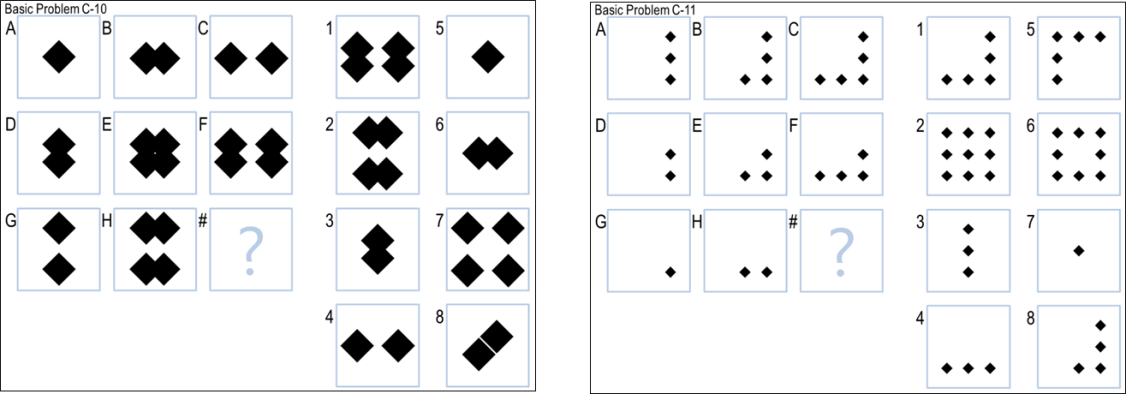


**Figure 9**

To solve these “single-object” problems, I include into my agent a new transformation named “number of objects increased by # fold”. First I iterate through A-H to make sure that they are all composed of the same object (with the same shape, size and fill). Next, I simply compared the fold of change in terms of object counts. In Basic ProblemC-03, E🡪F has the transformation “number of small filled diamonds increased by 1.500 fold”. When applied to panel H, it helps to find the best answer #4. Similarly, Basic Problem C-04 can also be solved by this strategy.

2) **Variations of “single-object” problems**

For these “single-object” problems, sometimes there is more than one choice with the correct number of the object. For example in Basic Problem C-10 (**Figure 10- left**), #1, #2 and #7 all fits the criteria of “number of filled diamond increased by 2 fold” when considering the subset A-C-G-#. In this case, to differentiate these choices, my agent needs to take other positional attributes into consideration (such as overlaps, above, left-of), and excludes the overlapped #1 and #2, leaving #7 as the best answer.



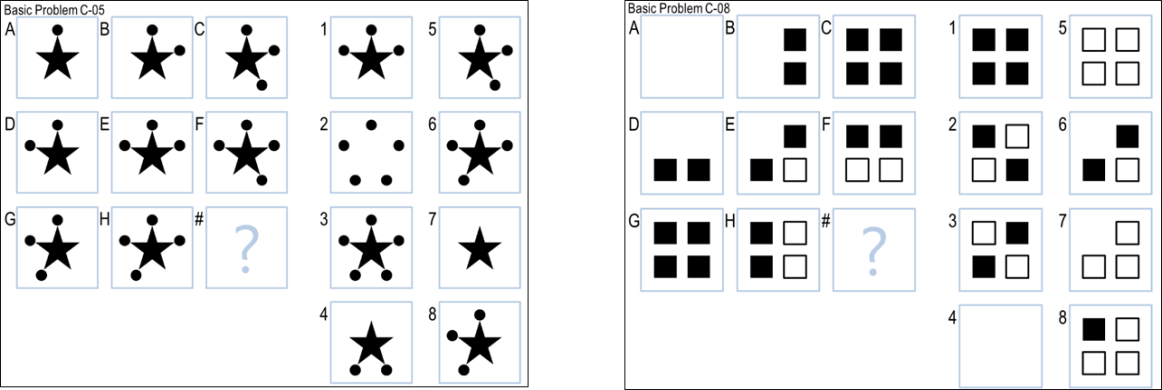
**Figure 10**

Another variation for the “single-object” problems is the counts of the object is not necessarily changed by fold (arithmetic progression), they can also change by arithmetic sequence. For example, in **Figure 10- right**, when considering the 2x2 subset “G-H-H-#”, the number of small diamonds increases by 1 (instead of 2 folds) each time. Thus #3 and #4 can both be the answer. Next, when taking the positional attribute “above” into consideration, my agent is able to exclude #3 and pick up #4.

**3) Multi-objects problems**

Although some problems do not belong to the “single-object” problems, as they contain multiple objects with different basic attributes, the idea of “change of number” can still be applied. For example, in Basic Problem C-05 (**Figure 11**), although panel A-C contains two kinds of objects (a star and some small circles), the star remains the same in all 8 panels as background! Thus I modified my agent to group objects into “unchanged” and “changed in counts” categories and ignore objects that always stay as background. This algorithm helps me to correctly solve Basic Problem C-05.

However, for a similar situation in Basic Problem C-08 (considering the 2x2 subset E-F-H-#, filled squares are background and unfilled squares changes in number), my agent didn’t get it correctly. Through debugging I figured out that the agent failed to ignore the filled squares in panel E, F and H, probably because the filled squares are very similar with unfilled squares considering all the other attributes. Thus the matching of objects did not go as I expected. Since the correct matching of objects between panels is fundamental to my algorithm, although tweaking the matching criteria could favor this particular problem, it becomes overfitting and affects the results of a few other Raven problems. Thus I failed to solve the Basic Problem C-08.



**Figure 11**

**4) When to skip a problem**

In project 2, a penalty has been introduced for wrong answers. So it is not the best strategy to always return the answer with the lowest score of difference (a variable I keep track of to compare answers). I went through the score for all RPM problems solved by my agent, and realized that for all the problems my agent has a reasonable algorithm to respond, the scores of difference for the best answer are always <= 5, and for those problems I don’t have a reasonable rational, the scores are greater than 6. Thus, I set a threshold to be 6 and only accept answers with a score <=5. If the score of difference is greater than 6, I consider answer is picked by “guessing”, thus rather skip it by returning “-1”.

5) Some **edge cases** I failed to consider in project1

For example, a RavenFigure may not contain any RavenObjects, namely, a specific panel could be empty. I need handle this situation to avoid null pointer exception that crushes my code. Additionally, when iterating to compare object for the “single-object” problems, some objects may not have certain attributes (size, fill, or neither), but should still be considered as the same object. Thus null value should be taken care of before comparison.

With all the improvements I made, I can get **10 correct answers out of 12 RPM problems** (1 skipped, 1 incorrect) in Basic Problems B in a split of second.

**V. Problems remained**

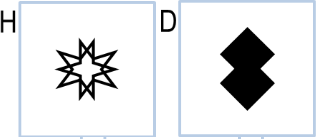
**1. Attributes about relative position**

Although I can solve most of the RPM problems successfully in Basic Problems B and C, I can still foresee some potential caveats if more complex problems are encountered. One of such problems is: in the current version of solve(), I treat positional attributes as "yes" or "no" questions for simplicity. For example, for attribute "left-of: b" in Basic Problem C-09, I only consider whether the object "a" is left of anything or not. Although this simple algorithm is sufficient in solving this problem, part of the original information is apparently lost, as this attribute points to another RavenObject "b". Pointing to another frame or instance is too complicated to consider for now. However, one can easily imaging that a specific type of RPM problems can be designed to trick my current version of Solve() into unstable or wrong answers.

**2. Verbal presentation**

The Raven problems are represented visually as well as verbally. For this project, I still used verbal representation. Although I don’t have to worry about how to convert image information into Raven objects, the verbal representation has its own problems and limitations. One of the problems for verbal representation is the **loss of information** compared to visual representation. For example, in the txt files, the size of a object is usually described as “small”, “medium”, “big”, “very big” and “huge”. However, what if there is an object appeared with the size in between “small” and “medium”? Since “size” is a continuous variable, which potentially has infinite categories, the verbal representation will eventually run out of description or becomes too complicated to interpret. However, the visual representation can convert size into numerical measurement.

Another example, as shown in **Figure 12**, both panels have attributes describing the two objects as “overlaps”. However, “overlaps” is not accurate enough to differentiate these two relationships: the two stars overlap with their center, but the diamonds only partially overlap on their edges. Generally, this is a problem one can hardly get away with when comes to irreversible data compression. Thus, to improve the performance of current agent, one has to work on ways to extract information from the visual representation.



**Figure 12**

**VI. About human cognition**

Sometimes, we human beings can solve RPM problems with hidden rules we don't even realize, a behavior somewhat like intuition. For example, we glimpse the answers, and pay attentions to the most likely answer first. The computers can only iterate with a fixed order. Additionally, we constantly make logic deductions, generate hypothesis about correlations then find evidence to confirm them. We gained these abilities most by receiving all kinds of stimuli from the world. Our memory reinforces the experiences that we can often find stereotypes in the environment, thus facilitate us to better adapt to our lives. Thus the responses can be both helping and illusive, but will not affect the reasoning of our computers.

For example, for Basic Problem C-07 (**Figure 13**), when we examined the eight panels together on the left, we easily consider this as a “moving” transformation: the small black dot moves towards the center and then away from the center. So if we have this hypothesis in mind, when we look at panel E, we see the small black dots overlaps with the small diamonds in the center. However, for an AI agent, regardless verbal or visual representation, reads panel E as a single black dot, thus considers the transformation to be diamond appeared/disappeared. For this particular problem, the agent still chooses #2 as its answer due to the other hints. However, one can appreciate the huge differences in terms of understating between artificial and real intelligence.

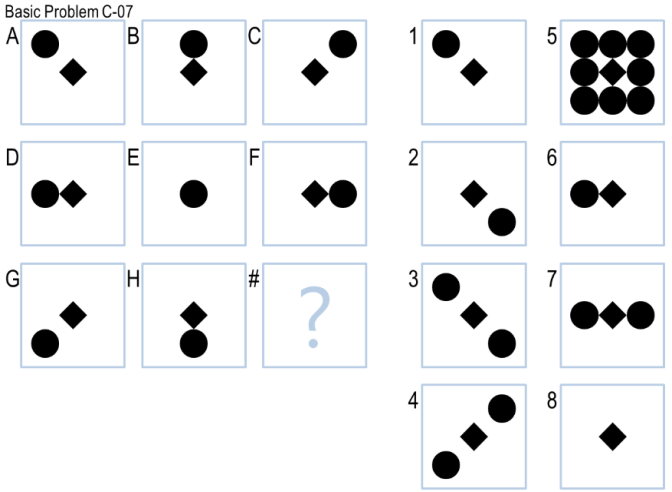


Figure 13