**Solving Raven’s Progressive Matrices problems with visual methods**

Project 3 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 1, my agent can solve 2x2 RPM problems with verbal presentations. In project 2, starting with the simpler model built in project 1, based on the new requirements and problems encountered, I improve my AI agent to solve more complicated 3x3 RPM problems, but still with verbal presentations. Here, in project 3, I implemented new methods to solve RPM problems solely based on their visual representations.

**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 🡪 ProblemSets 🡪 Individual Raven Problems 🡪 Raven Figures

* 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 D” or “Challenge Problem D”) and a list of Raven problems.
* Individual Raven Problem is represented by **RavensProblem**.java, which consists of a problem name (such as “Basic Problem D-01”), a problem type (such as “2x2” or “3x3”) and a HashMap storing different panels of RavenFigures.
* Each **RavenFigure** is composed of a figure name (such as “A”-“H” for questions, or “1” -“8” for answers), a string variable visualFilename storing the path of a visual file. The class RavenFigure also contains a HashMap of Raven objects, but I won’t be using this information for my visual method this time.

With the path of a .png image file, my agent reads in the visual presentation of a RavenFigure:

RavensFigure A = problem.getFigures().get("A");

BufferedImage figureAImage = ImageIO.read(new File(A.getVisual()));

In my agent, I implemented a method called convertTo2DUsingGetRGB() to covert data stored in the BufferedImage into an int 2D array:

int[][] A2D = convertTo2DUsingGetRGB(figureAImage);

In this method, I iterate every pixel of this BufferedImage through each row and column, and get the RGB value. Since all the RavenFigures are supposed to be greyscale images, I simply averaged the blue, green and red values and ignore the alpha value. In the converted int 2D array, white pixels are represented by “0” and black pixels are represented by “1”. The tricky question is how about the grey pixels? I decided to define a grey pixel with an RGB value smaller than a threshold (say 200) to be a black pixel and greater that 200 to be a white pixel.

private int[][] convertTo2DUsingGetRGB(BufferedImage figureAImage) {

int width = figureAImage.getWidth();

int height = figureAImage.getHeight();

int[][] result = new int[height][width];

for (int row = 0; row < height; row++) {

for (int col = 0; col < width; col++) {

int color = figureAImage.getRGB(row, col);

int blue = color & 0xff;

int green = (color & 0xff00) >> 8;

int red = (color & 0xff0000) >> 16;

result[row][col] = (255-(blue + green + red)/3+200)/255;

}

}

return result;

}

**III. Core algorithms**

In my agent, I mainly implemented two core algorithms: set operations and counting pixels. With these two algorithms combined, I was able to solve most of the RPM problems in Basic Problem set D and E with **23 correct answers out of 24 RPM problems** (1 skipped) in a few seconds.

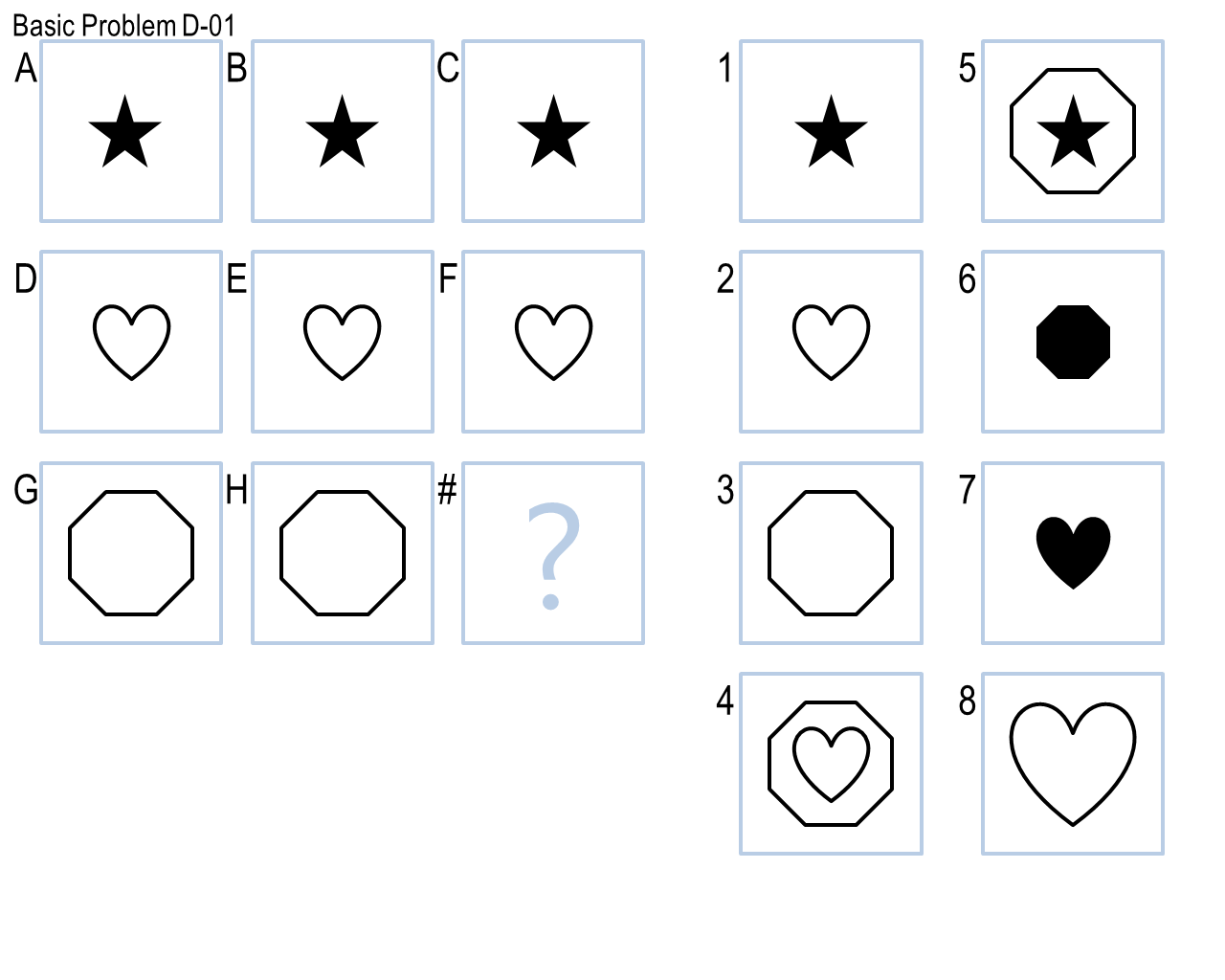
|  |  |  |
| --- | --- | --- |
| **Problem** | **Agent's Answer** | **Correct?** |
| Basic Problem D-01 | 3 | Correct |
| Basic Problem D-02 | 1 | Correct |
| Basic Problem D-03 | 3 | Correct |
| Basic Problem D-04 | 1 | Correct |
| Basic Problem D-05 | 7 | Correct |
| Basic Problem D-06 | 1 | Correct |
| Basic Problem D-07 | 1 | Correct |
| Basic Problem D-08 | 4 | Correct |
| Basic Problem D-09 | 3 | Correct |
| Basic Problem D-10 | 1 | Correct |
| Basic Problem D-11 | 3 | Correct |
| Basic Problem D-12 | 3 | Correct |
| Basic Problem E-01 | 1 | Correct |
| Basic Problem E-02 | 7 | Correct |
| Basic Problem E-03 | 2 | Correct |
| Basic Problem E-04 | 8 | Correct |
| Basic Problem E-05 | 5 | Correct |
| Basic Problem E-06 | 8 | Correct |
| Basic Problem E-07 | 3 | Correct |
| Basic Problem E-08 | 1 | Correct |
| Basic Problem E-09 | -1 | Skipped |
| Basic Problem E-10 | 8 | Correct |
| Basic Problem E-11 | 5 | Correct |
| Basic Problem E-12 | 6 | Correct |

1. **Identical figures**

For some simple RPMs, the figures in each row seem to be identical, as shown in **Figure 1**. I implemented a method called computeDiff() to compare pixel-by-pixel the differences between two 2D arrays representing their RavenFigures and return the number of different pixels in int.

private int computeDiff(int[][] a, int[][] b)

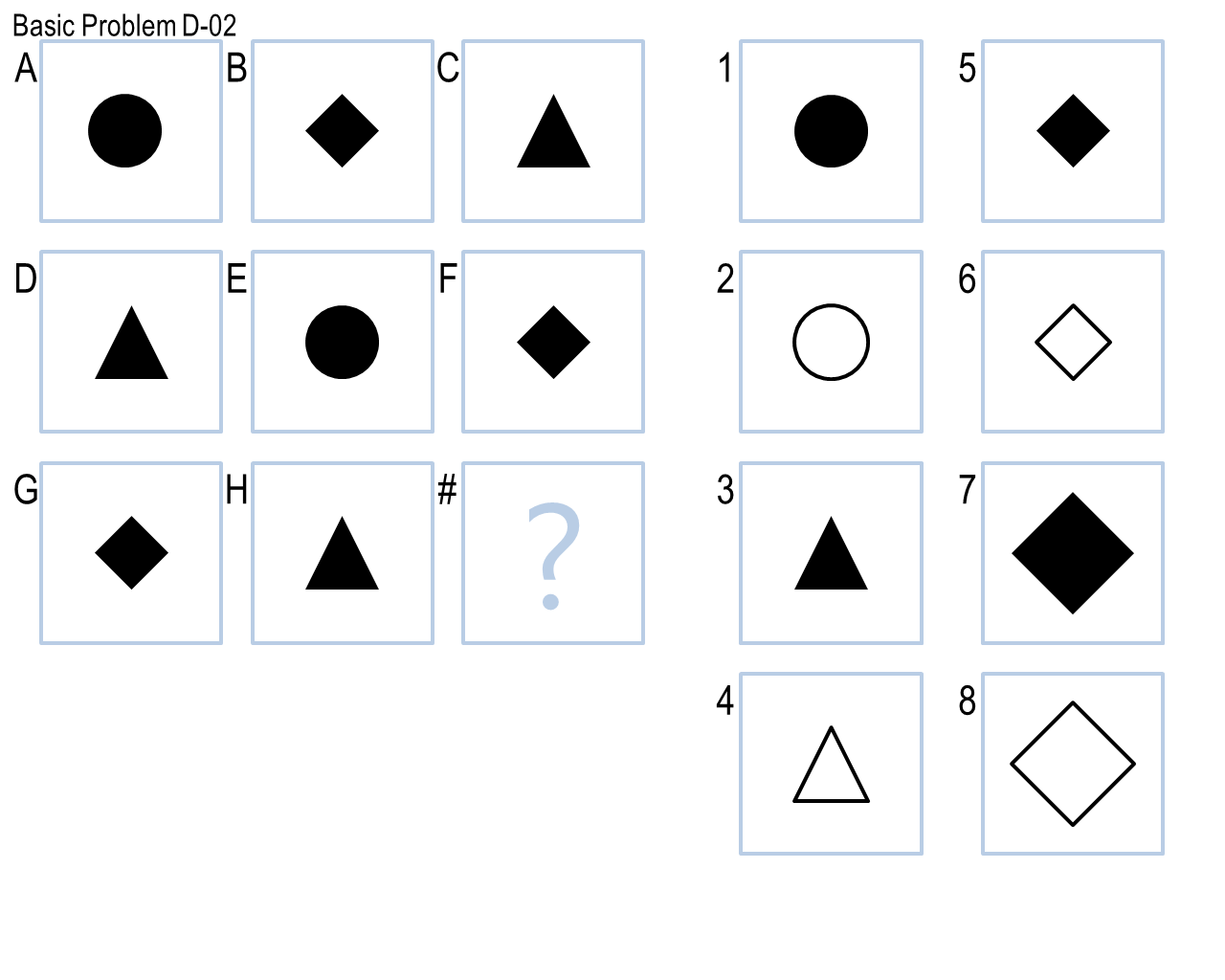
Theoretically, computeDiff (A, B) should return 0. However, since these figures are not necessarily copies of each other, they may have slight differences that could hardly noticed by human eyes. I set a threshold, say as long as computeDiff(A, B) is less than 20 pixels, I will consider A and B to be identical figures.

****

**Figure1**

If a certain problem satisfies my agent’s criteria (A is identical to B and C, D is identical to E and F), the transformation is labeled as “**identical row ABC**”. Then the agent will generate an answer based on panel G and H, and pick up the best matching answers from choices 1-8.

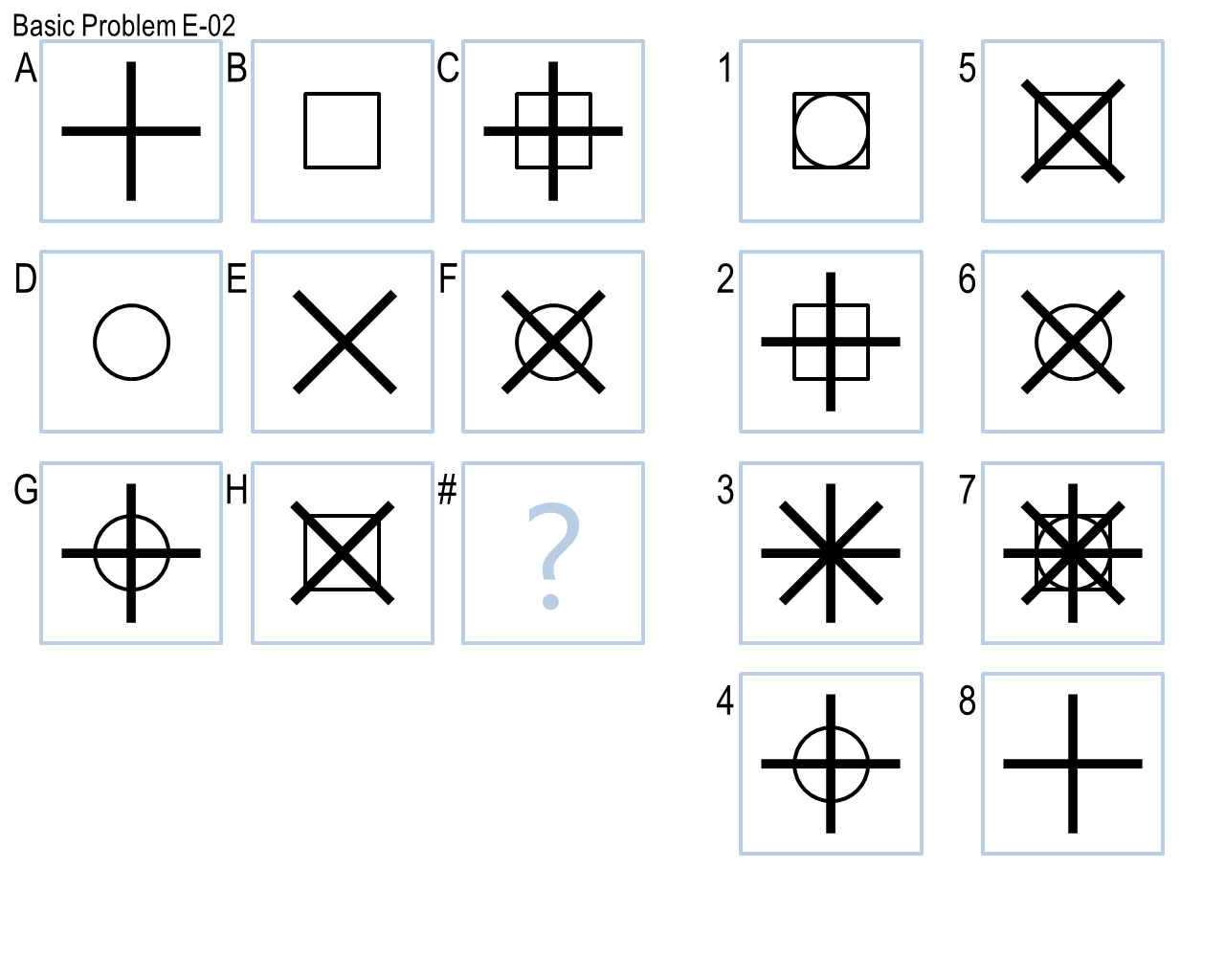
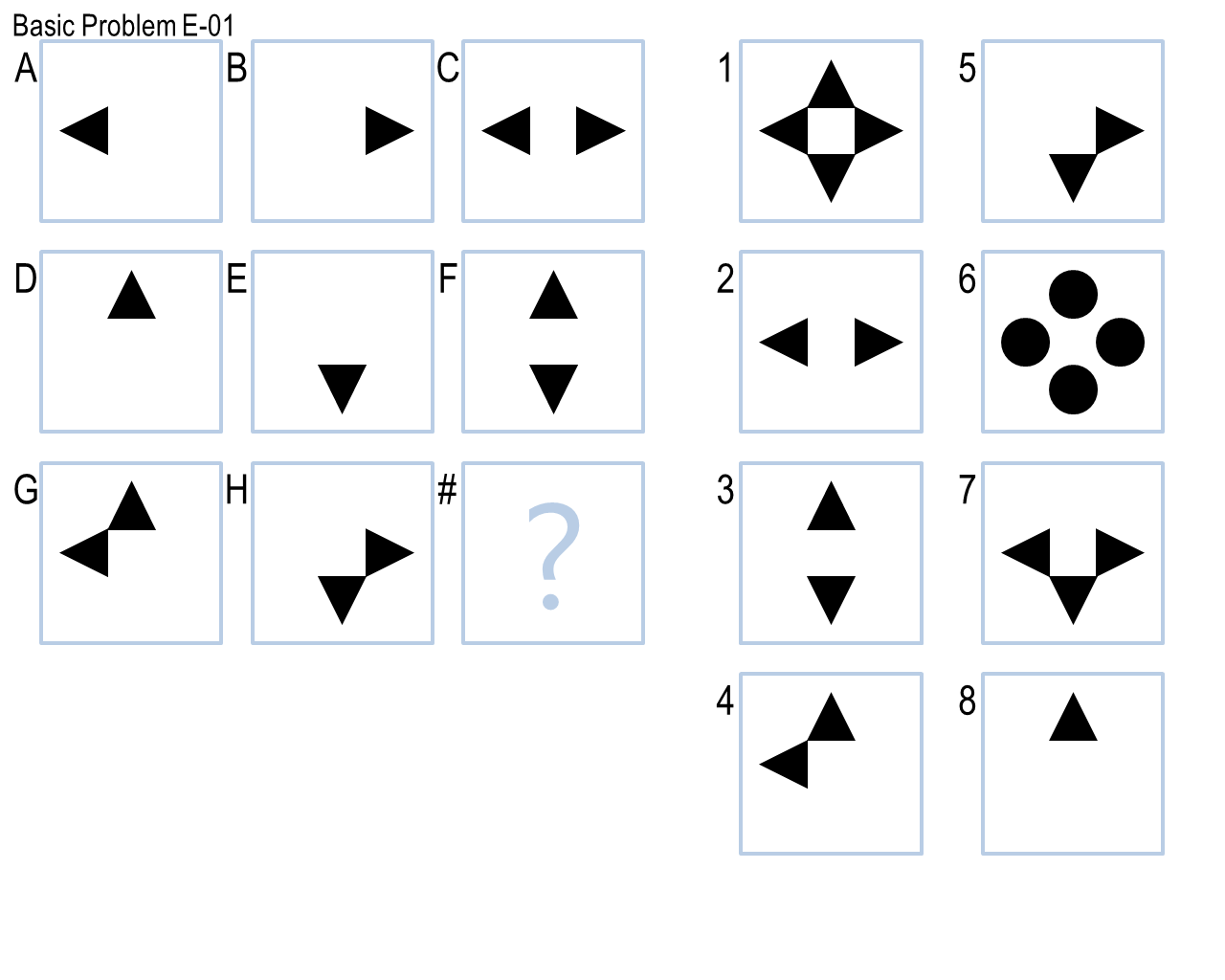
“Identical Figures” problems can have **variations**. For example, in **Figure 2**, the identical figures appeared diagonally instead of horizontally. In this case, the criteria becomes “B is identical to F and G, D is identical to C and G” and the transformation will be labeled as “**identical diagonally BFG**”. Then the agent will generate an answer based on panel A and E. Other similar variations are also expected, I will not enumerate all of them.



**Figure 2**

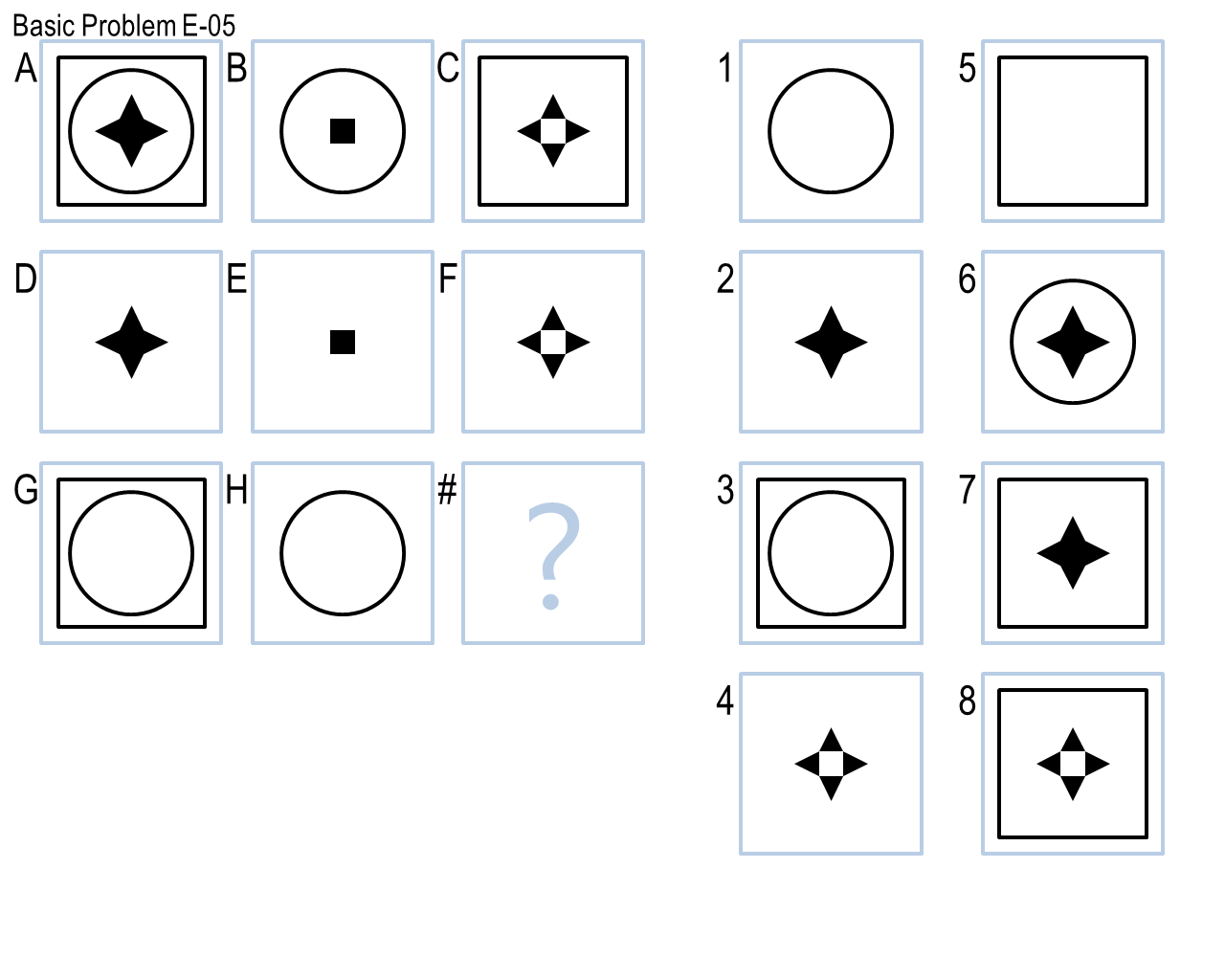
1. **Set operations**

Quite a few problems can be solved by set operations: Addition, subtraction, union, intersection, XOR. For example, in **Figure 3**, overlapping of panel A and B gives panel C, and overlapping of panel D and E gives panel F. So my agent defined this transformation to be “**UnionABC**”. It is expected that overlapping of panel G and H will give the answer for this question. So the agent will generate an answer based on panel G and H, and compare it to choices 1-8. . Notice that there may be some variations of problems sharing the same bottom-line algorithm (such as A U C ==B or A U D ==G), and then we can define similar yet different transformation such as “**UnionACB**” or “**UnionADG**”.



**Figure 3 (Union example)**

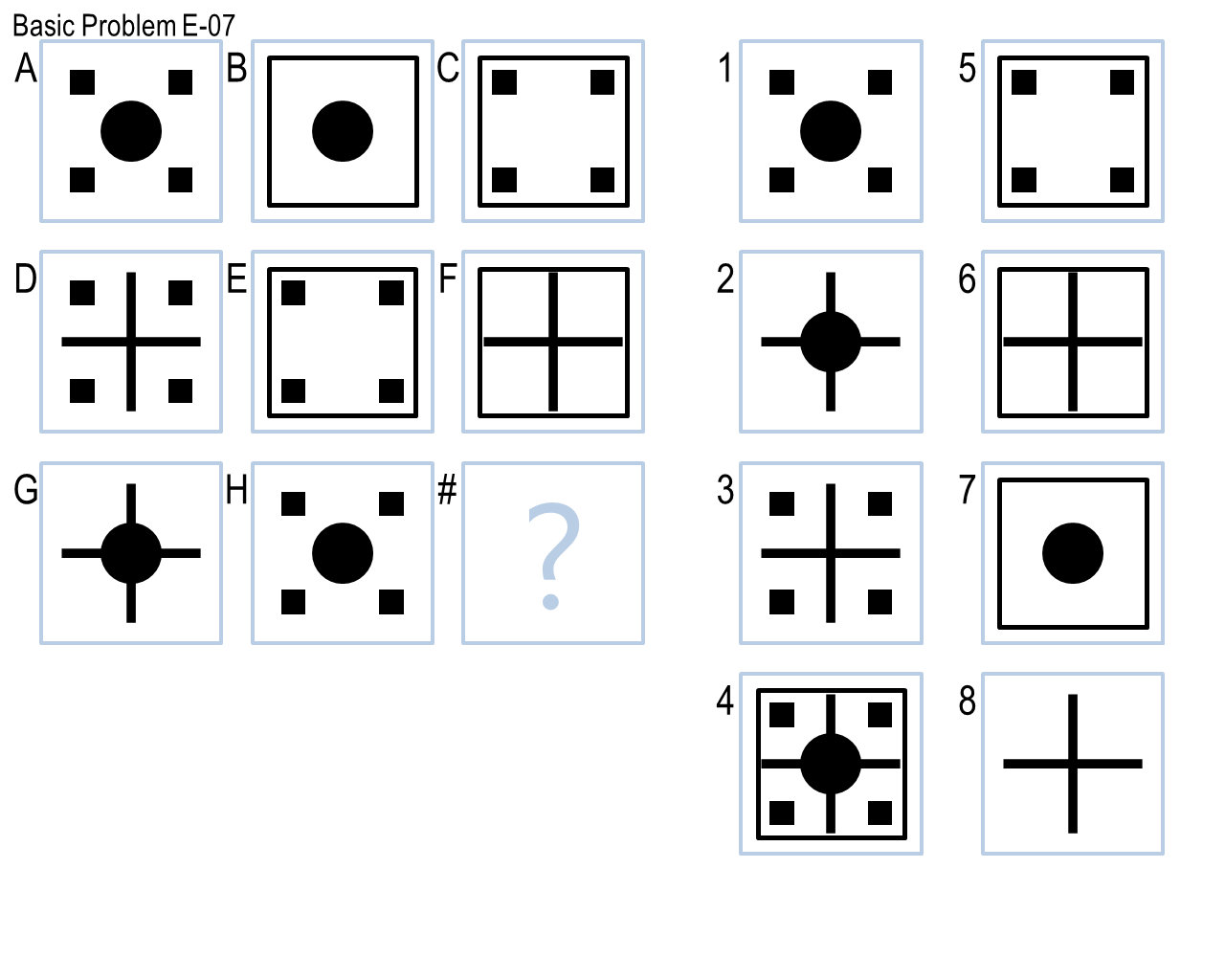
**Figure 4** is an example of the **subtraction** transformation, as subtracting B from A makes panel C, and subtracting D from E makes panel F. So my agent defined this transformation to be “**SubtractABC**”. The expected answer can be generated by subtracting H from G.



**Figure 4 (Subtraction example)**

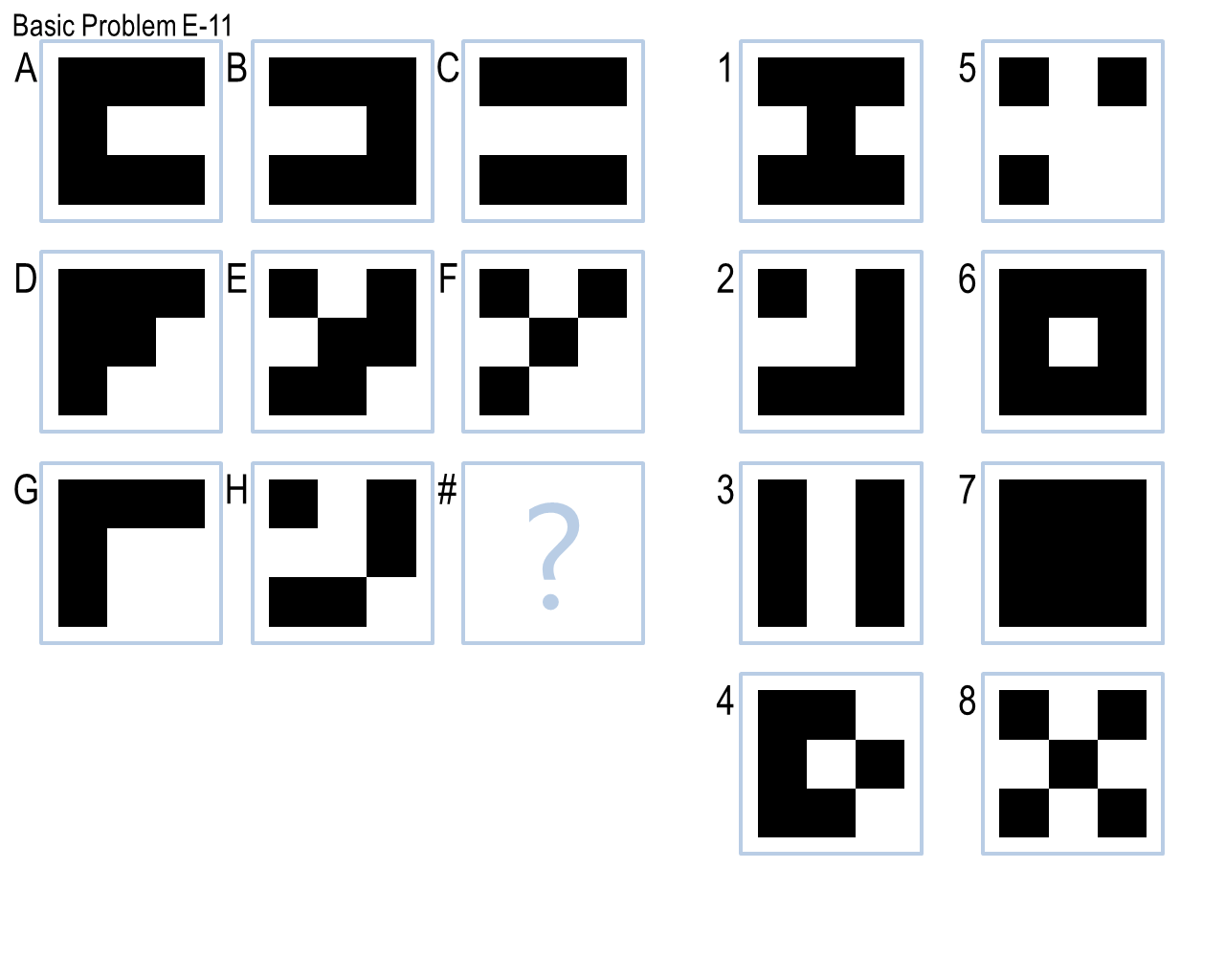
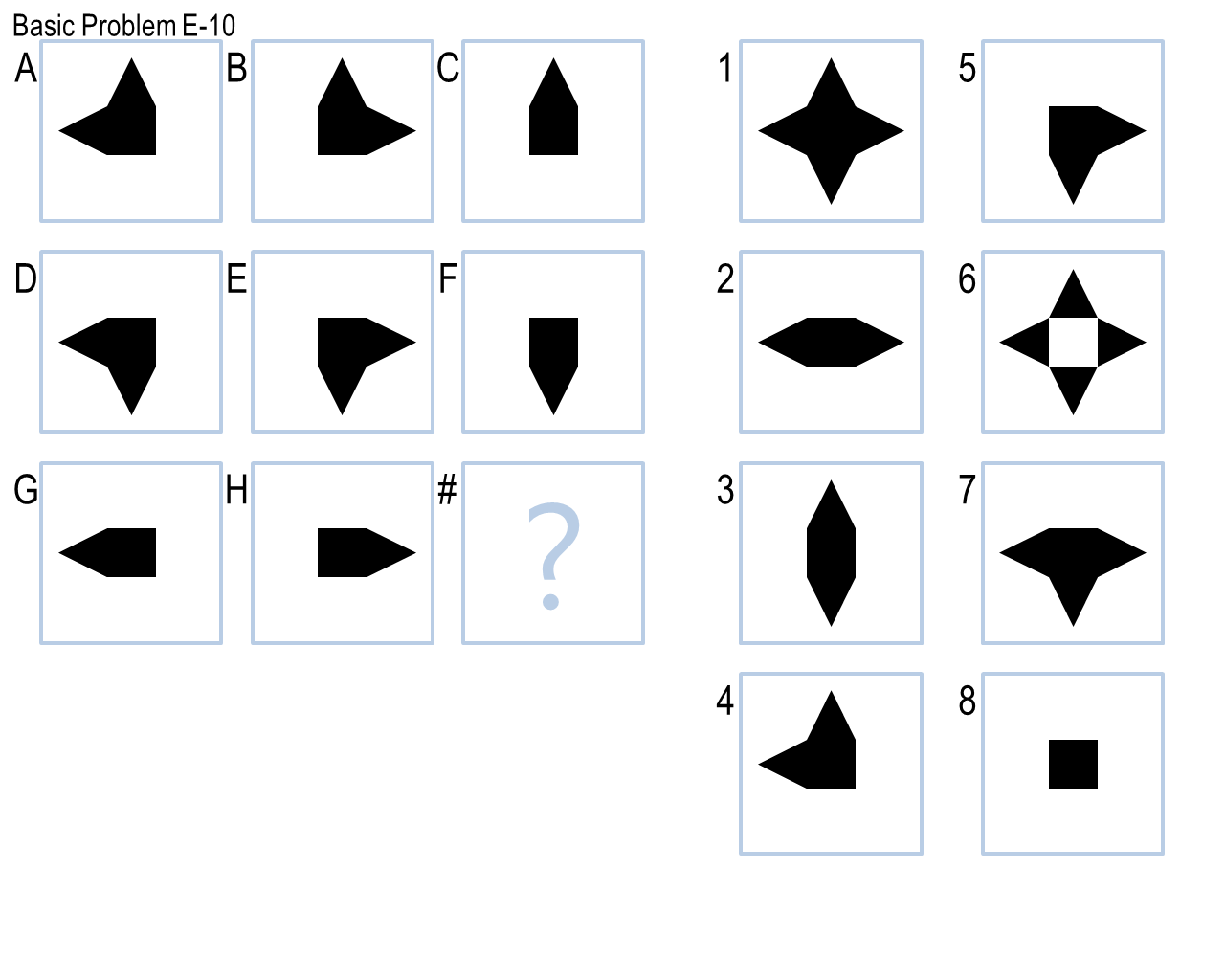
Similarly, my agent also includes transformation using other set operations, such as:

“XORABC”: A **XOR** B 🡪 C as observed in figure 5.



**Figure 5 (XOR example)**

“IntersectABC”: A **intersect** B 🡪 C as observed in figure 6.

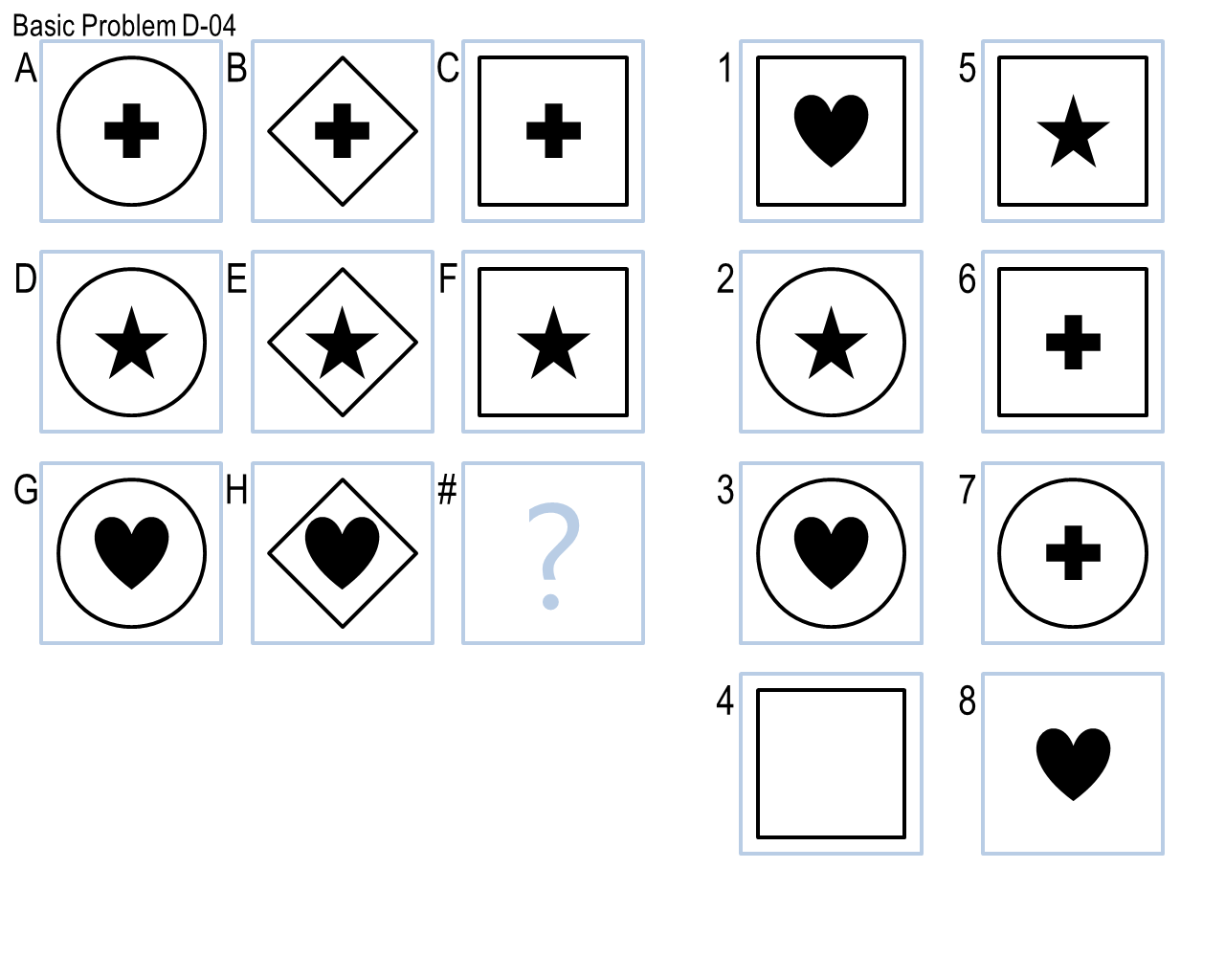
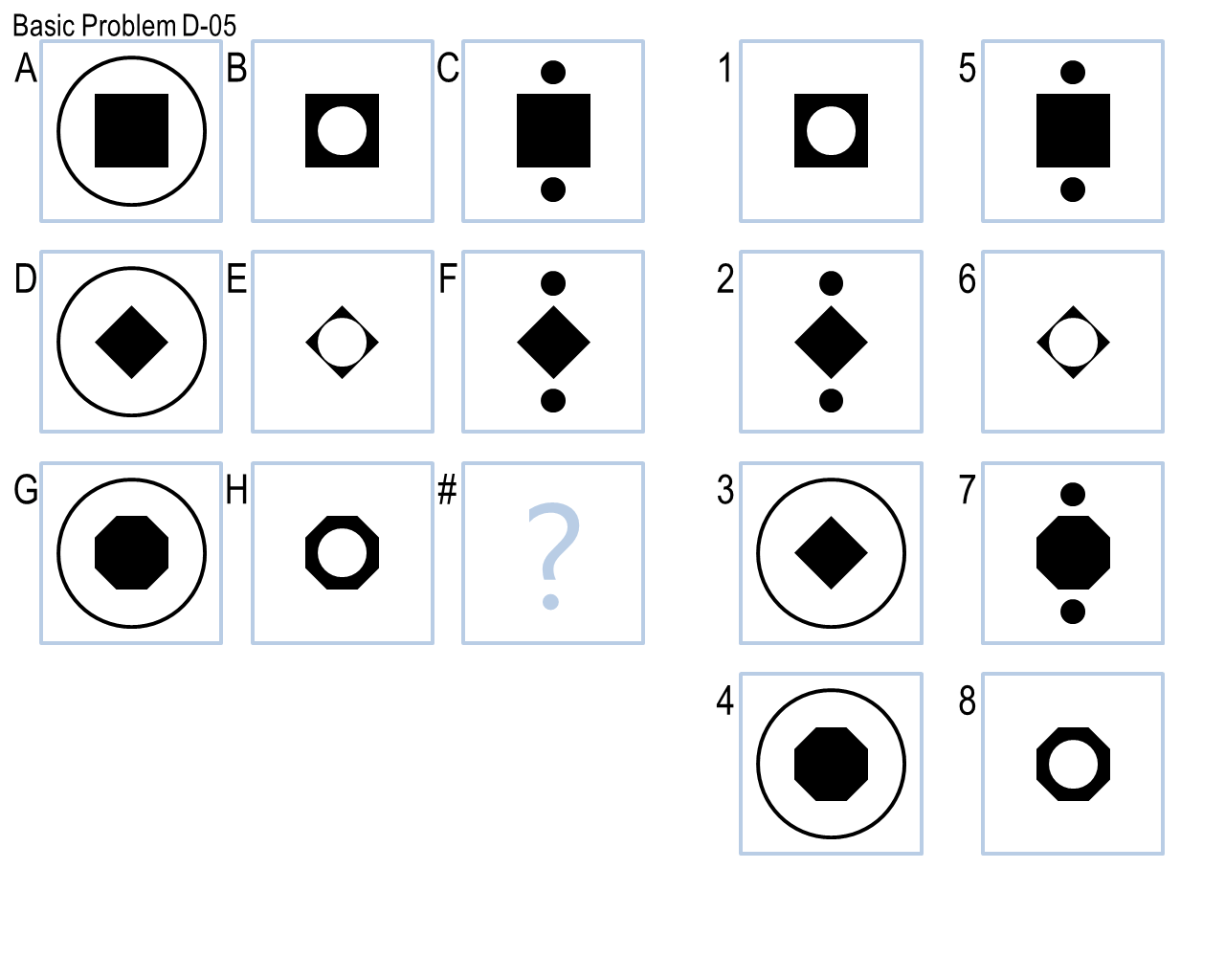


**Figure 6 (Intersect example)**

1. **Counting Pixel**

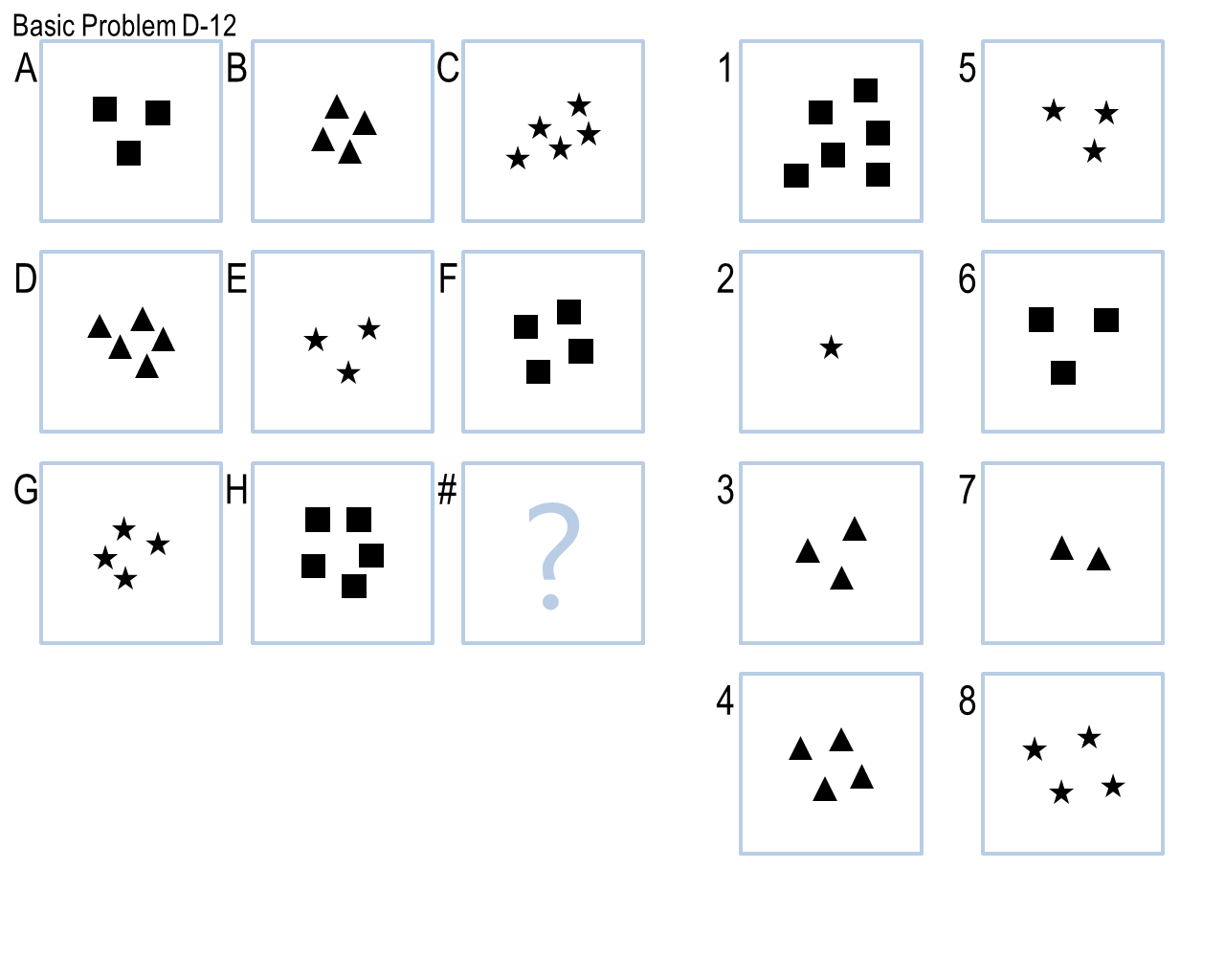
Although not quite intuitive for human cognition, computer can actually solve a lot of RPM problems by simply counting the number of black pixels in each figure and make connections. For example, **Figure 7** is a combination of two “identical figures” type of problems: the inner object is identical horizontally, while the outer boundary is identical vertically. Normally, we human beings will separate each panel into two objects and find a combination of transformations that matches changes in both directions. However, my AI agent does not yet have the capacity to separate figures into individual object, but it can still solve this problem. When my agent compares the pixel differences between A and B and C, the inner identical object will be ignored, and only the differences related to the outer object matters.

For problems in **figure 7 and 8**, my agent found that pix(A)-pix(C) == pix(D)-pix(F), thus define this transformation to be “**pix\_ACDF**” (considering slight shifts of some figures, small differences should also be allowed). Then it calculates the expected pixel numbers based on pix(G) and the difference pix(A)-pix(C), and then examined which choice answer has the closest number of pixels.

****

**Figure 7 Figure 8**

Sometimes, the **fold change** rather than the actual number of pixels is playing a role. For example, in **Figure 9**, pixels ratios in A : F : H and E : G : C are both 3:4:5. Then the pixel number expected can be calculated based panel B, panel D and the changes of fold. Although lots of information is lost, the counting pixel method is still very powerful to solve lots of simple RPM problems.



**Figure 9**

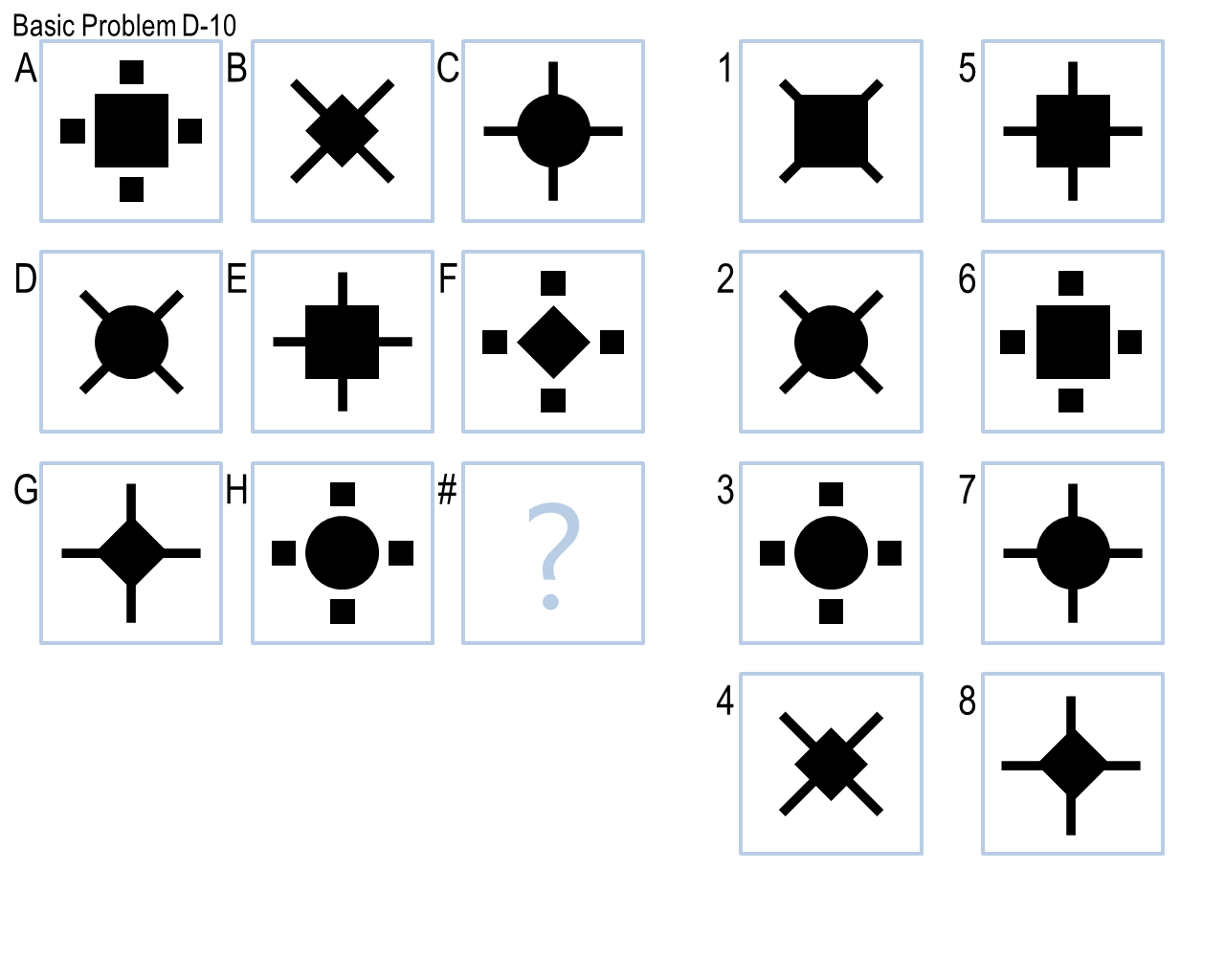
1. **Combine Set Operations and Counting Pixel**

Some problems can be solved by combining the set operation transformations and the counting pixel problems. For example, the problem shown in **figure 10** is very similar to the one shown in Figure 7. The change of pixel number in F🡪H🡪A and G🡪C🡪E and B🡪D🡪? should be the same and it should be solved as a diagonal pix count problem. However, for the answer shown in #1, the number of pixel is smaller than the expected pixel number by calculation, as the solid square in the middle is partially overlapping with the cross (see red region in Figure 7). So an improved solution for this problem is:

1) Using counting pixel number method to define the transformation to be “AcoverH”

2) Using set operations to pick up an answer that can completely cover panel D (answer U panel D == answer D), but different than panel D (answer U != panel D).

This method was specifically designed to solve this particular RPM problem, and show that it is useful to combine multiple algorithms to solve some more complicated problems.



**Figure 10 (example of combining algorithms)**

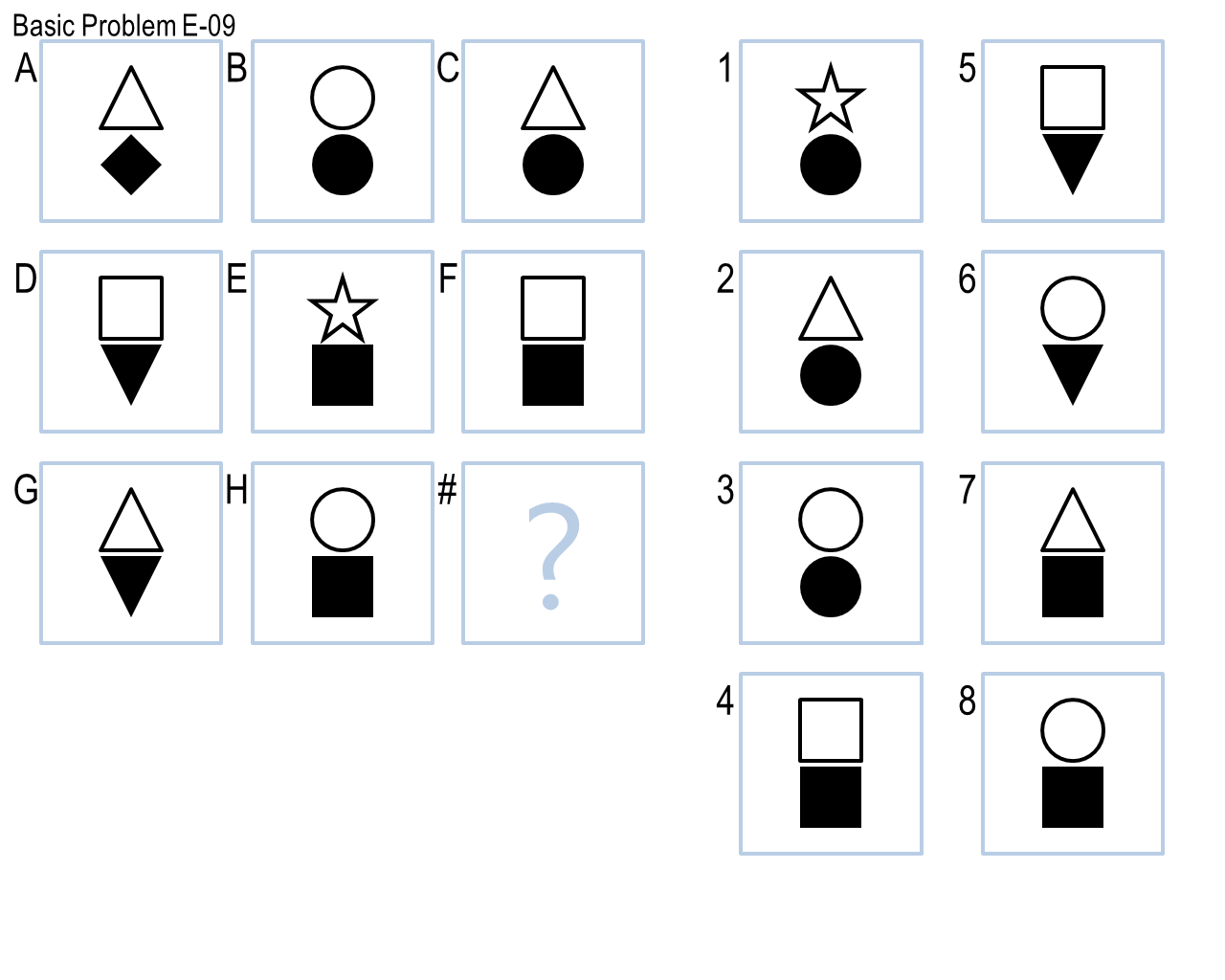
1. **How to pick up the best answer?**

For RPM problems that can be solved with set operation algorithm, the expected answer is usually generated from the known panels and the detected transformation. Then all the candidate answers will be compared to the expected answer, and the most similar one will be returned as the best answer. However, for RPM problems with transformations that do not fall into any category of set operations, my agent will then try the count pixel algorithm to find correlations in terms of pixel count. Then the expected pixel count of the answer will be calculated and compared to those of the candidate answers. Often, narrowing the scopes of candidate answers by ruling out unlikely/identical figures can greatly reduce noises and increase the chance of picking the correct answers. Finally, if the problem does not fit into any transformation implemented in the agent, it will be skipped by returning -1.

**IV. Problem remained**

My agent approves to be a successful trial for my first implementation of visual method. However, there are still many problems and gaps, which could potentially help to broaden and improve this original frame in the future. For example, for the kind of problems shown in **figure 11**, which I called “**shirt and pants matching problem**”, will have to be split into two objects (bottom and up). The transformations of these two parts are independent (like a person wearing different shirts and pants), and need to be considered separately. Similarly, a Raven problem can be composed of left and one right object, multiple objects at the four corners, and etc. The combination lists can go on and on, and the objects can change independently or somewhat cooperatively. A comprehensive agent needs to include all possibilities to be able to generalize its solution to all kinds of problems.

Additionally, beside the algorithms described here, the agent also needs to be able to divide panels into individual objects and be able to reason like how it did with the verbal presentations eventually. This will be a much more advanced RPM solving agent, and should be the direction of future improvements.

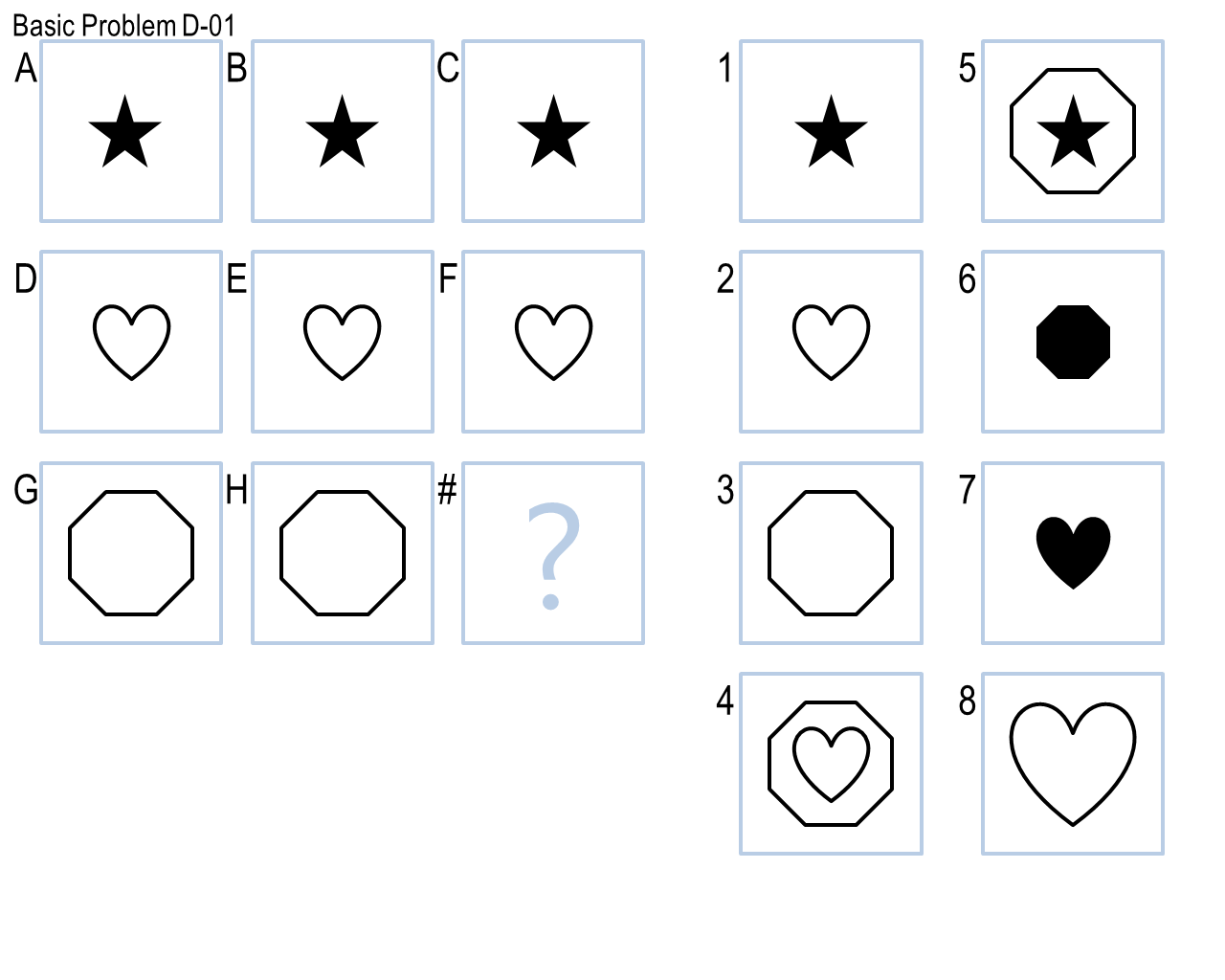


**Figure 11 (shirt and pants matching problem)**

**V. Related to Human cognition**

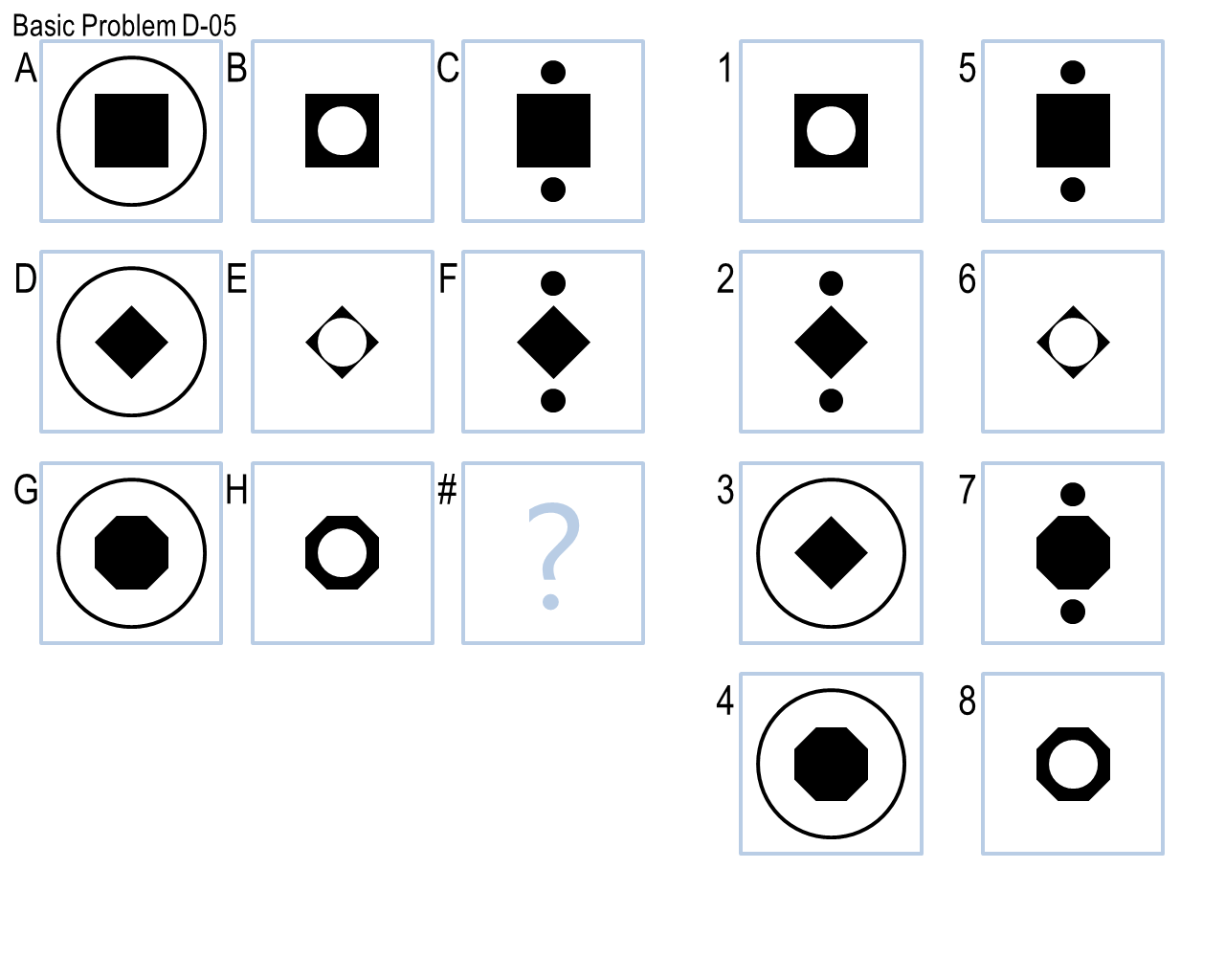
Sometimes, we human beings can solve RPM problems with hidden rules we don't even realize, a behavior somewhat like intuition. For example, we glimpse the answers, and pay attentions to the most likely answer first. The computers can only iterate with a fixed order. Additionally, we constantly make logic 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 helpful and illusive, but will not affect the reasoning of our computers.

For example, we consider panels in figure 12 are identical, as by every definition we used to examine the world (such as shape, size, fill, color, position), they are the same. However, when AI read in these pictures as 2D array, it fails to group this problem into an “unchanged” transformation, as these panels will be considered different even though there is only a slight shift, unless artificially defined under certain conditions with a threshold set in advance.

****

**Figure 11**

On the contrary, it is not very likely for human being to solve RPM problems in Figure 12 by counting pixels. One would naturally start to divide panels into objects and try to find correlations. However, it is so much easier for AI to solve purely based on pixel counting. Actually, counting the pixels is such a straightforward and reliable method for AI and should always be given high priority in trying to solve RPM problems. Additionally, imaging a huge (for instance, 10x10) RPM problem, most people will have problem to remember the information from those 100 panels, let alone find correlations in them. However, memory of this size is hardly a problem for AI if the transformation algorithm remains the same complexity level. With these examples, one can at least appreciate the huge differences in terms of understating between artificial and real intelligence.

****

**Figure 12**