Project 1 reflections

**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. 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? In this project, I used knowledge learned in the past lectures and implemented an AI agent to solve the 2x2 RPM problems.

**Data structure**

First I decided to go over the data structure provided in the provided starting files and add necessary classes. Overall, the problems are represented by "Frames" or classes in OOP. 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 (a class storing problems of the same difficulty level). The ProblemSet is composed of a ProblemSet name (such as “Basic Problem B”) and a list of Raven problems. Individual Raven Problem is represented by RavensProblem.java, which consists of problem name (such as “Basic Problem B-01”), problem type (such as “2x2” or “3x3”), a HashMap storing different panels of figures. The Raven problems are represented visually as well as verbally. For this project, I am going to simply extract information from the verbal representation. In later projects, I need to figure out ways to extract information solely from the visual representation. Each figure in a Raven problem is composed of a figure name (such as “A”, “B”, “1” or “6”), and a HashMap of Raven objects. Finally a Raven object 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”).

In order to present transformations from one figure to another, **I constructed a new class named Transformation.java** in Agent.java. This class contains two RavenObjects, named source and target, storing the source figure and target figure of this transformation. This class also has a list of string named mutations, storing each mutation (such as “expanded”, ”rotate” and “reshape”) in this particular transformation. They are all private fields, which can be accessed and updated by their getters and setters method.

**public** **class** Transformation {

**private** RavensObject source;

**private** RavensObject target;

ArrayList<String> mutations;

//getters and setters

}

**Method implemented**

In the Solve() method, I first read in information of one RavenProblem, and point each figure to a RavenFigure instance:

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

Then I implemented in new method called computeTransformation() to compute the transformation between two figures:

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

I use this method to compute the transformation between Figure A and B, as well as transformations between Figure C and all the candidate answers. Then I implemented another method to compute differences between two transformation:

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

Next, I use this method to evaluate how similar these 6 candidate transformations are compared to the expected transformation from A to B. In the end, I iterate to pick up the candidate answer with the best score and set it to be the given answer of the problem.

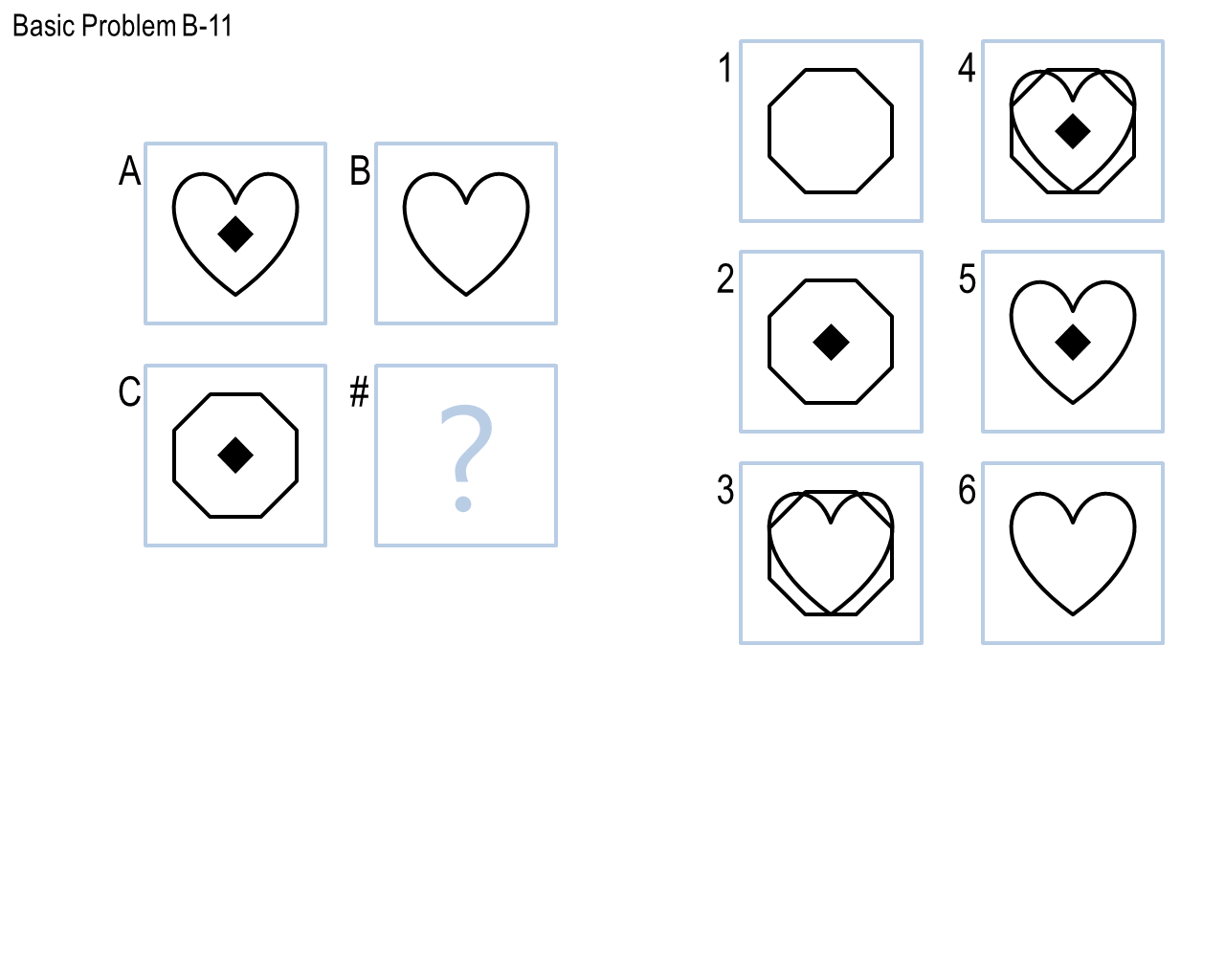
**Improvements during implementation**

One of the biggest obstacles when I was implementing computeTransformation() is: **how to pair up objects from two figures** or say, how to match the corresponding object in another figure. My original thought was to compare objects appeared in the order from textual representation, but soon realized it is not optimal, because naming and order of objects in one particular figure can be totally random. Thus, my solution is to use two "for loops", going thought all possible object pairs between figure A and B, and find the best match between objects. The "best match" is evaluated by a method named getDiff() to quantify the difference between two objects:

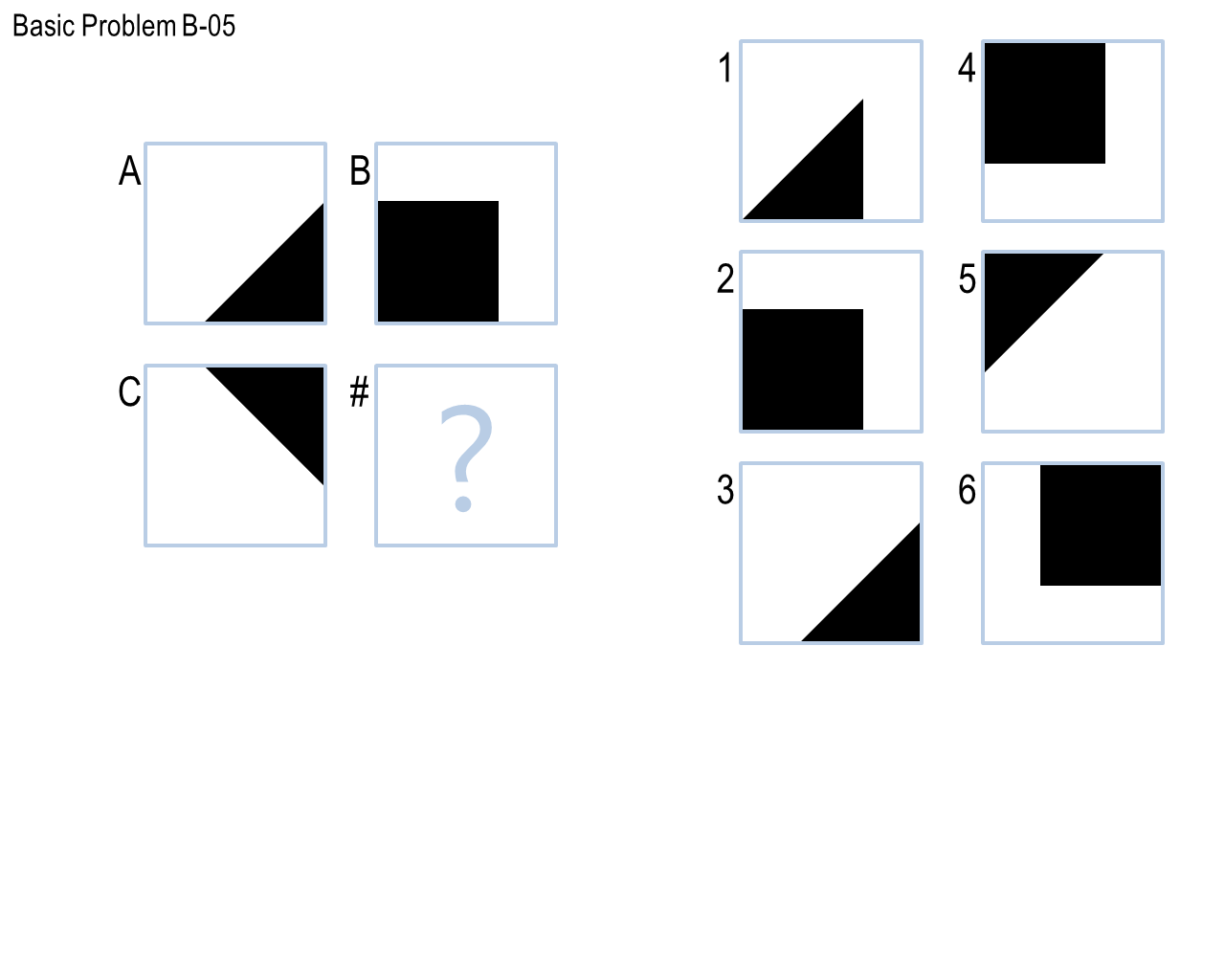
**private** **int** getDiff(RavensObject ob1, RavensObject ob2)

Similarly, this getDiff() method uses two iterations to find the best match between attributes, and return the lowest difference between RavenObjects in forms of integer.

Some **edge cases** need to be considered here. For example, if one of the objects appear/disappear in a figure during transformation, some object may not find a corresponding one in the other figure and will be overlooked. In this case, I have a HaspMap to keep records of whether all the objects are matched, and if not, penalty scores will be added. In an older version of my code, the first object in an iteration always found a matching object no matter unreasonable it might be and this can be a problem. For example in Basic Problem B-11 as shown below, the first of object ended up disappeared (the diamond) and disrupted the matching of other remaining objects (heart and octagon). Thus, my solution is set a threshold of min\_Diff = 2, which means, if the differences between two objects are greater than 3 attributes, I refuse to match them up. This has a little bit of "hard-coding", but is indeed efficient in the current situation to keep noise down.



To generate a transformation between two figures, I need to define ways to **convert the changes in attributes to some transformation behavior**. For example, in Basic Problem B-11, the transformation between A and B can be defined as "**disappear**" (or " diamond disappear" to be more specific). For the changes in size, I defined a HashMap to map attributes such as "big", "medium" or "small" to integers, so that I can directly compare these size attribute values and assign strings "**expanded**" or "**shrunk**" to the mutation arraylist. If the shape attributes do not match, I can add "**reshape**" or "reshape from shape1 to shape2" to the list. If the fill status changes, I can append "**filled** changed from yes to no" or " filled changed from no to yes " to the list. For **alignment** attribute, I described the mutation as "alignment changed from bottom-right to bottom-left". In this case, "bottom-right" and "bottom-right" and "bottom-left" are information extracted from the verbally representation. However, in Basic Problem B-05, from A to B, one of the mutations is "alignment changed right to left" rather than "alignment changed from bottom-right to bottom-left". My current solution is to add "alignment changed right to left" to my mutation list. However, this brings up a general question: how to generalize these mutations, and in the meantime keep it specific enough to narrow down the best choice, or in another word: how to balance generalization and specification. This is one problem I am still working on. One strategy is to implement another evaluation method to assign intermediate penalty scores to similar but the same transformation.



So far, the solve() has been treating the 2x2 RPM problems as 2x1 problems. To upgrade the solve() to be able to **solve the 2x2 problems**. I simply call my previous procedures once again to compute the transformation between A and C, and also the transformations between B and all the candidate answers. 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. With all the improvements I made, I can get all **12 correct answers out of 12 RPM problems** in Basic Problems B!

**Problems remained**

Although I can solve all 12 RPM problems successfully in Basic Problems B, 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 "a inside:b", I only consider whether the object "a" is inside of anything or not. Part of the original information is apparent 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.

**About human cognition**

Sometimes, we human beings can solve RPM problems with hidden rules we don't even realize. 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 can make logic deductions, make hypothesis then find evidence to confirm them, and we often switch from one to the other back and forth or integrate them together to solve a problem. This behavior may be referred this as "intuition", an ability we start to gain, when we were born and started to receive information from the world, or even when our ancestors were created and tried to survive within the wildness. Some reactions have been encrypted into our genes and inherited. These are responses we can often find stereotypes in the nature, thus facilitate us to react better to our living environment. Thus they can be both helping and illusive, but will not affect the reasoning of our computers. For example, in Basic Problem B-04, my solve() originally gave me #1 as the correct answers, as it considered the transformation A--> B to be "rotating 270o", which exactly converted figure C to #1. Thus it is correct by definition. However, there is another possible transformation from A to B: vertical mirroring and this is a transformation more easily picked up by human eyes. Thus, I implemented the criteria for vertical mirroring, namely (angle1+angle2) %180==0, and added the string "vertical mirroring" in my mutation list. Finally, I set the priority of "vertical mirroring" (and also horizontal mirroring) higher than the one of rotating, so that in the way computer "learn" how to think like a human being.

