Seth Henneman
henneman@gatech.edu

Project 1 Design Report

The goal of Project 1 is to develop a KBAI agent capable of solving 2x1 Raven's Progressive Matrices. In order to succeed, our agent will require a suitable mechanism for representing and storing information contained within the problems. It will also need a method for applying its knowledge to actually solve the problems. In our case, our agent will utilize semantic networks as the knowledge representation scheme. It will implement a combination of generate & test and production rules to perform problem solving. In the following sections, we will take a deeper dive into the design of our AI agent.

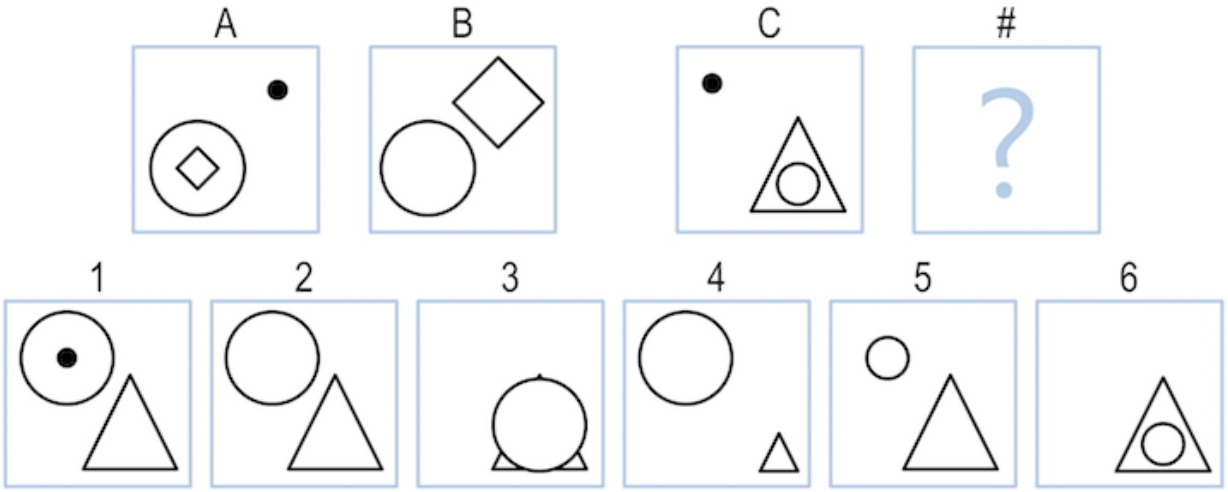

Figure 1. Example 2x1 Raven's Problem

Semantic Networks

Semantic networks will be the knowledge representation vehicle for our agent. These networks consist of 3 basic properties: lexical unit, structure, and semantics. The lexical unit, or node, in this case will be each shape that exists in a Raven's figure. The information stored in these nodes contain properties describing the shape itself. An example of node representation is shown in Figure 2.

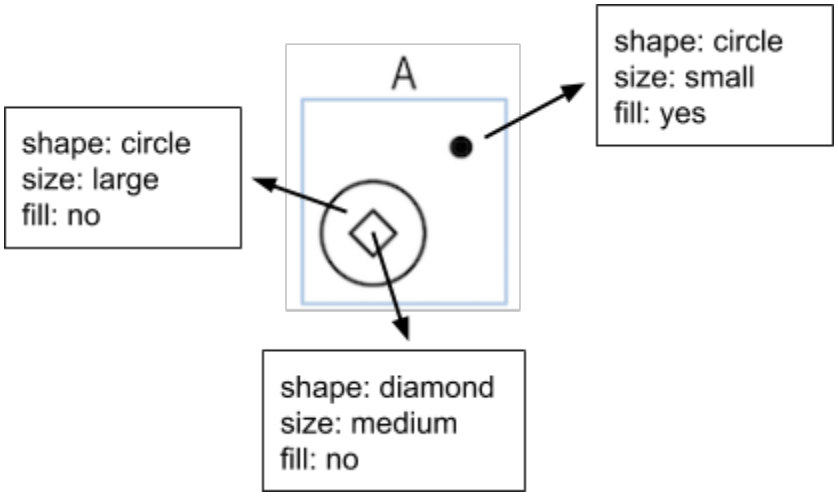

Figure 2. Semantic network node representation

Our agent, which uses a Java implementation, defines classes to store the nodes of our semantic networks. The classes used for this are RavensObject, which identifies the nodes, and RavensAttribute which stores all the properties associated with each node.

The structure of our semantic networks will consist of logical links between nodes, both within a single figure, and nodes that are correlated across figures. One of the most important jobs our agent has is to correlate nodes, or shapes in this case, between associated figures. In the case of 2x1 matrices, this means the agent is correlating each shape between figures A and B, and figure C and each answer. This important step is setting our agent up for the process of generating the transform information which represents the semantics property of our network.

Smart Generator

Our agent implements a smart generator for two purposes:

1. To correlate shapes between associated figures
2. To generate and store transform information between the correlated shapes

The smart generator uses a production rule system to correlate shapes between figures. These production rules define a set of preconditions that will trigger a node correlation if the preconditions are met. It turns out that a very simple set of production rules were able to correctly correlate objects in all of the test problems presented to our agent. The correlation production rules are:

Compare shape x in figure 1, to each shape y in figure 2:
If shape matches, then
(1-a) If size or fill matches, then trigger correlation (shape AND size|fill matches)
(1-b) else trigger correlation (shape only matches)
(1-c) else shape was either added or deleted

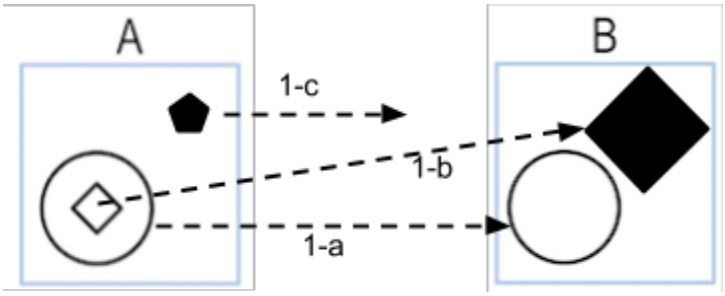

Figure 3. Smart generator node correlation

There are scenarios where multiple correlations are possible. One useful mechanism for accounting for this is to alter the order in which our agent attempts to correlate objects. In this case, different answers may be generated which can be compared to find the best one.

Once our smart generator has correlated the objects between figures, it can generate the transform information which will ultimately be used by our smart tester. Our agent once again utilizes a production rule system to trigger the types of transforms that exist for each shape. There are too many production rules implemented to list them all here, but here are a few examples:

If object exists in figure 1, but not in figure 2, then transform=DELETED
If object contains "inside" property in figure 1, but not in figure 2,
then transform=MOVED_OUTSIDE
If object size in figure 1 is "small" or "medium", and size in figure 2 is "large",
then transform=EXPANDED
If object angle in figure 1 is not equal to object angle in figure 2,
then transform=ROTATED

Our agent implements a RavensTransform class to capture all the transform information required for the smart tester to do its job. This class contains a simple integer bitmap (shown below), as well as other variables to store information which can't be suitably captured by the bitmap (such as rotation degrees). Keeping this data structure as simple as possible is important for the performance of our agent.

```
/**
 * Transform Bitmap
 * This bitmap stores the transforms between correlated nodes in related figures.
 * If a transform is present between the correlated nodes, the associated bit is set.
 */
public static final int UNCHANGED = 0;
public static final int DELETED = 1;
public static final int ADDED = (1 << 1);
public static final int SHRUNK = (1 << 2);
public static final int EXPANDED = (1 << 3);
public static final int FILL_CHANGED = (1 << 4);
public static final int MOVED_INSIDE = (1 << 5);
public static final int MOVED_OUTSIDE = (1 << 6);
public static final int MOVED_ABOVE = (1 << 7);
public static final int MOVED_BELOW = (1 << 8);
public static final int MOVED_LEFT_OF = (1 << 9);
public static final int MOVED_RIGHT_OF = (1 << 10);
public static final int FLIPPED_VERTICALLY = (1 << 11);
public static final int FLIPPED_HORIZONTALLY = (1 << 12);
public static final int ROTATED = (1 << 13);
```

Smart Tester

Now that our agent's smart generator has captured all the correlation and transformation information, the smart tester will compare the set of transformations associated with figures A and B to the transformations between C and each possible answer. The smart tester implements a similarity measure to gauge which possible answer's transformation set is closest to that of figures A and B.

The similarity score system utilized by the smart tester will award points if certain comparison conditions are met. For example, if the number of transformations (other than UNCHANGED) between figure C and the possible answer matches the number of transformations between figure A and B, then the agent will award two points to the possible answer. There are several other comparisons being made as well, such as whether or not the transform bitmap matches, or whether there are shape, size, fill, or rotation matches.

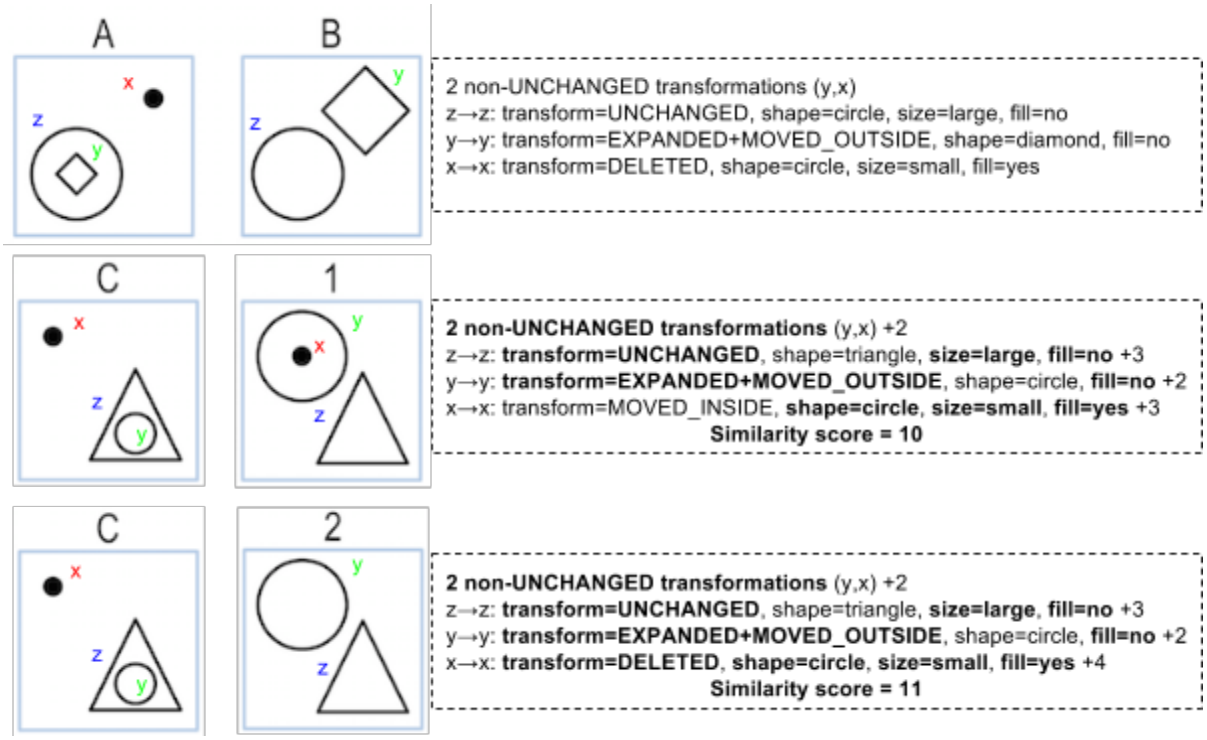

Figure 4. Smart tester similarity score example

Once the smart tester has calculated a similarity score for every answer, the answer with the highest value will be submitted by the agent as the answer for the given Raven's problem.

Results

When we applied our agent's implementation to the set of basic problems provided, the agent correctly answered 20 out of 20 problems. It was also able to answer the 2 classmates problems correctly. Unfortunately, it was not successful at answering any of the challenge problems, which would require further design considerations. One example of a challenge problem our agent unsuccessfully answered is shown below.

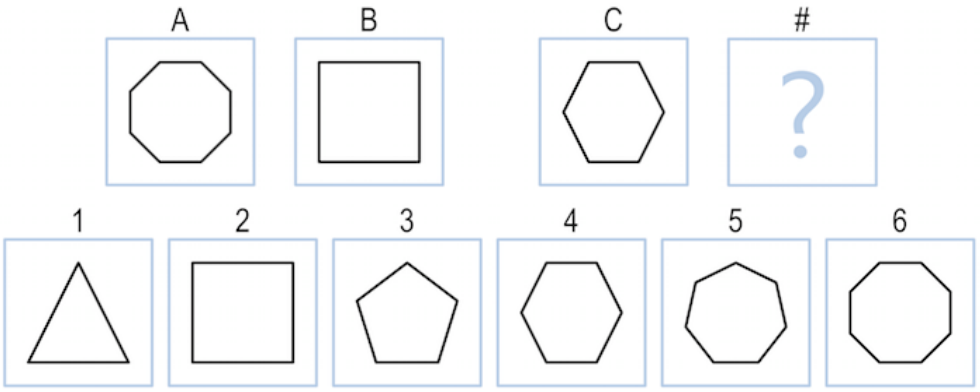

Figure 5. Sample Challenge Problem

One current limitation of our agent is being able to calculate the number of sides of a given shape. This functionality would be required in order to successfully address problems like this one.

Had we been given more time to develop our agent, there is certainly room for improvement. One major consideration would be to implement more generic methods for addressing not only 2x1 Ravens problems, but also 2x2 and 3x3 problems. Another consideration would be to abstract some of the methods out more to account for new properties our agent may not have seen during testing. It's also possible our agent could implement a mechanism for "learning from its mistakes". This would make our KBAI agent more human-like.