Actions:

action('keep', shape_id)
Ensures the shape indicated by shape_id does not disappear from the first diagram to the second.

action('shrink', shape_id)
The shape defined by shape_id becomes a smaller size in the second diagram.

action('enlarge', shape_id)
The shape defined by shape_id becomes a larger size in the second diagram.

action('moveLeft', shape_id, dist)
The shape defined by shape_id moves left by a distance of abs(dist) in the translation from the first diagram to the second.

action('moveRight', shape_id, dist)
The shape defined by shape_id moves right by a distance of abs(dist) in the translation from the first diagram to the second.

action('moveUp', shape_id, dist)
The shape defined by shape_id moves up by a distance of abs(dist) in the translation from the first diagram to the second.

action('moveDown', shape_id, dist)
The shape defined by shape_id moves down by a distance of abs(dist) in the translation from the first diagram to the second.

Heuristics:

Prior to delving into the exact heuristics I used to draw conclusions, I will outline the data structures I created to make the heuristics easier to implement. My main agent was contained in a class that I called "Brain", this class would handle all of the thinking and answering that the agent would need to do. In order for it to draw any meaningful conclusions, however, it needed to divide the information it received into different categories. As such, I also created a class that represented a shape, a diagram, and an interpretation. In order to remember and manipulate all of this data, I needed to create a data storage class that held the information on every shape the agent came across. I called this the "Subconscious Memory". Nested inside of the Brain were instances of Diagram::Diagram and Interpretation::Interpretation. This allowed the agent to freely associate the information located in each of those classes. The SubconsciousMemory::SubconsciousMemory object, however, was kept out of the Brain class so that the data inside of it would not be deleted accidentally upon stack-frame exits.

Once the data had been read and organized, I designed my heuristic to work in all cases. This way, I would not have to identify an analogy problem as a different subclass of problem, etc. Instead, all analogy problems were solved using the same basic logic. In English, the big picture was:

1. Match the shapes in B with the similar shapes in A.
2. Match the shapes in C with the analogous shapes in A.

3. Determine what changed in each matched A shape to each matched B shape from the transition from A->B.
4. Predict an answer by applying the transforms of A->B to the matched shapes in C. Save this hypothetical diagram.
5. Determine which K diagram has an interpretation most similar to our predicted diagram.
6. Return this K.

In this explanation, I use the words “Similar” and “Analogous” to mean separate and inverse things. Both words “similar” and “analogous” are defined on a scale from (0,1]. For every difference in the physical qualities of the shape, it’s “similarity” or “analogy” decreases. Therefore, if a shape did not change in any way from Diagram1 to Diagram2, both its similarity **and** analogy would be 1. However, the agent considered similarity and analogy to be different relationships when ranking 2 shapes.

A **similar** shape is thought to be a shape that retains its shape **type** (triangle, square, circle, etc.), but simply changes location or size. Despite this, if 2 shapes are vastly far apart in location, they are considered **less** similar than 2 shapes that are close by.

An **analogous** shape is thought to be a shape that has a similar location or size, but instead changes its type. Similarly, if 2 shapes are very far apart, they are considered less analogous than 2 shapes that are closer in location.

To see these relationships in greater detail, see the “RankSimilarity” and “RankAnalogy” functions in my code.

In addition to ranking similar and analogous shapes, the agent needed to rank the similarity and analogy of interpretations. This was achieved by averaging the similarity or analogy ranks of every shape contained in the interpretation. As a result, the ranking system was still on a scale of (0,1]. This had to be modified slightly by deducting rank if 2 interpretations had mismatching numbers of shapes.

As a result of this algorithm, my agent differs slightly from the agent that is hinted at in the spec document. My agent does not take into account shapes that were deleted from diagrams; instead, it simply doesn’t treat ‘delete’ as an action. It just ignores those shapes entirely. For the output, I created an implicit action to represent this: ‘keep’. ‘keep’ was not defined in the code prior to creating the output.

As a final note, I created this technique by asking friends and family to solve these puzzles themselves and then observing how their thought process went. I discovered early on that my test subjects seemed to be imagining and predicting a likely answer before considering the 3 choices. As such, I created an Agent that did just that.