Nathaniel Meyer (nmeyer7) nathaniel.meyer@gatech.edu

Project 1

My design goal for project 1 was to write an agent that could make decisions better than random guessing, and to be sufficiently general that there should be no loss of accuracy between the test and training sets. I chose the Java package, and use a Factory model to return the best solver for each Ravens problem. At this point, the factory logic returns the Project 1 agent for all 2x1 Ravens problems, and returns a random guesser for all others.

For my agent, I wanted to incorporate the concept that each problem has its own language, and it is important for the agent to understand both what is specified and what is not specified in each figure and object. To that end, my agent first learns the language of the problem by iterating over the descriptions of each figure and cataloging all of the different dimensions used by the describer, and their possible values.

In the process of learning the language, I realized that some information is leaked to the agent by what the describer chooses to describe, and chooses not to describe. If the agent also had to process the image to recognize features, shapes and characteristics, these problems would have had a great deal more information and it might not be in an immediately useful form. I took full advantage of the fact that a shape as a set of lines and vertices with edges and corners and fill that might be detected by an image processing algorithm was condensed to a shape and an angle by the describer. Additionally, my agent assumes that unimportant characteristics have been omitted. I try to mitigate this assumption later by taking into account all dimensions specified anywhere in the problem description for all objects when doing comparisons.

In order to solve the analogy in each problem, I needed a way to take the descriptions and come up with a testable hypothesis D that answered the relationship “if A is to B then C is to D”. Working backwards, my agent needed to learn how exactly A is to B before making any guesses about which D completes the C is to D relationship. I decided to use the langauge of the problem learned in the first phases and to iterate over all the dimensions for every object in each figure. ‘Absent’ is the value for any dimension not specified. Once a full description over all dimensions is generated, the agent excludes all dimensions that did not differ between A to B, and calls what remained a changeset.

The changeset represents a set of transformations, and each change in the changeset has a figure pair, an operation, a starting value and a resulting value. The figure pairs turned out to be another generate and test opportunity. I initially assumed that the object labels between figures were indicative of correspondence of those objects across figures, but I knew that was a weak assumption, and if it held true it was a serious leak of information by the describer. The assumption proved incorrect in at least one problem, so I abandoned it during development. To find the best mapping of objects in figure A to objects in figure B, I used the Steinhaus-Johnson-Trotter algorithm to generate all the permutations of possible relationships,

then I computed the changesets for each of them and scored the changesets on a complexity index similar to the one given in the course videos. The mapping of objects that produced the least complex changeset from A to B was selected, and the rest were discarded.
Future iterations of my agent might keep the top N least complex mappings, and solve the problem with each of them, deferring the choice of which mapping is the best until answer selection.

Having chosen a mapping of objects in figure A to figure B, and having computed a changeset representing the transformations from A to B (including deletion), I now needed a way to test whether this hypothesised set of transformations fit any combination of C to D that could be formed from the answers. I decided that the simplest way for my agent to decide which pair of figures formed the answer was to repeat the process done with A and B to come up with a changeset for each candidate C and D, and then to select the answer with the most similar hypothesized changesets.

To choose changesets, I had to come up with a way to score on the similarity of two changesets. I decided to add a point for each matching transformation operation, and another point if the start and end attributes of the transformation were the same. However, in order to properly score them, I not only needed to compare the list of transformations, I had to guess which object pairs in A/B corresponded to object pairs in C/D. I decided to have the agent use a similar routine to choosing pairs of objects in a single figure, and use the S-J-T algorithm again to generate all possible mappings of object pair relations, and compute the score for each mapping for each candidate answer. The agent then selects the best answer based on the highest scoring mapping of object pairs relations in A:B and C:D.

While I am pleased that my agent performed better than chance, getting 9/20 basic problems right, I am disappointed that I wrote a solver with only my own hard-coded intelligence, and did not find a way to feed back the error into the system to create a learning agent. My solution is a mathematical algorithm, entirely deterministic, with no capacity to learn. There are parameters which I could tune from the error given by CheckAnswer, but that would still only optimize within the frameworks of the hard-coded algorithm. I would have liked to design an agent that, upon getting a problem wrong, could try another strategy until it got the problem right, and could perform better on unseen problems for the practice in the current one. A learning agent, having seen a similar problem before, should be expected to perform better on future problems by virtue of that experience.   There is a lot of room for improvement in future versions.