Project 2 Reflection

Introduction

The path that I took in building an agent capable of solving the *Raven's Progressive matrix problems is to*[1] implement an intelligent generator, followed by an intelligent tester. There are several challenges that arose early on during the implementation process which will be discussed below; translating abstract theoretical ideas into practical applications is always a challenging task. Part of this challenge is due to the fact that limitations arise in terms of efficiency and data management.

NOTE TO GRADER ABOUT PROJECT 3 additions:
Content was updated for project 3 where a visual approach was taken. See relevant sections for updated strategy:

- 3x3 Problems Visual Approach
- Lessons learned from the 3x3 Visual Approach implementation

Approach

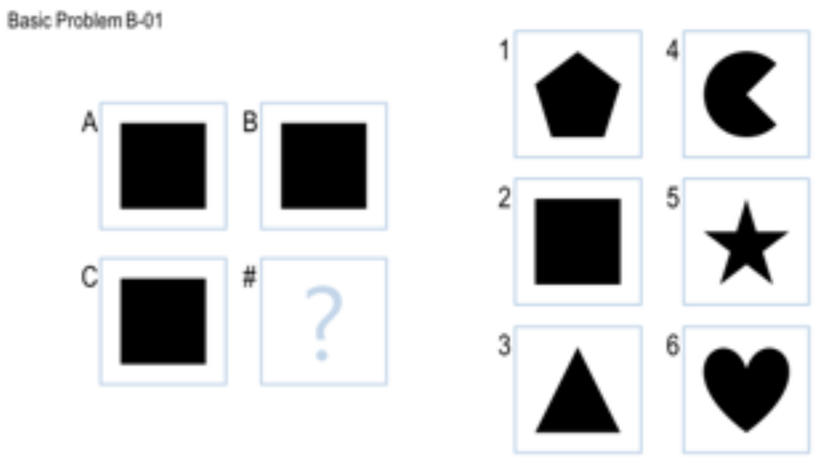
One of the first thing that I had to deal with right at the beginning was to determine how to manage the problem data. For example, The problem, as described in the *ProblemData.txt,* maintains a certain format. Therefore, several assumptions had to be made about the structure of these problem descriptions. Some of the assumptions that I made were as follows:

- Every single object described is unique
- There exist a set of known *reference object properties* (i.e "inside", "outside", "above")
- There exist a set of known alignment properties[2] (i.e "top-left", "bottom-left", etc)
- Objects that are rotated, are always rotated a common axis[3]

Making these assumptions, I have learned, yielded a fairly positive outcome for the agent.

Generator

As stated earlier, I have implemented an intelligent generator, in an attempt to generate the answer. In a very basic case, as in the case of *Basic Problem B-01,* The generator is very straight forward: Object a, maps to object b—therefore, the transformation that occurs between the two object is the same transformation that should occur between *C* and the answer.

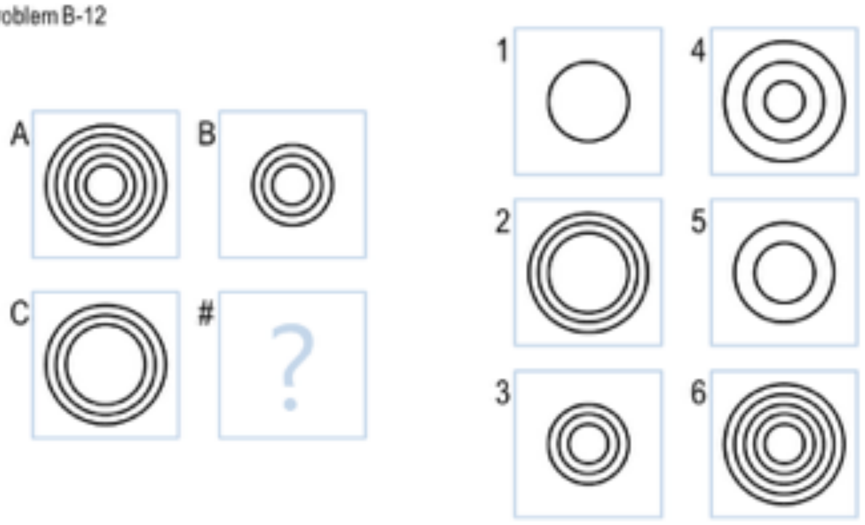


blems correctly

n due to rotations. This only
gures.

In order to challenge the generator, however, lets take a look at a more complicated problem, like *Basic Problem B-12*:

Basic Problem B-12



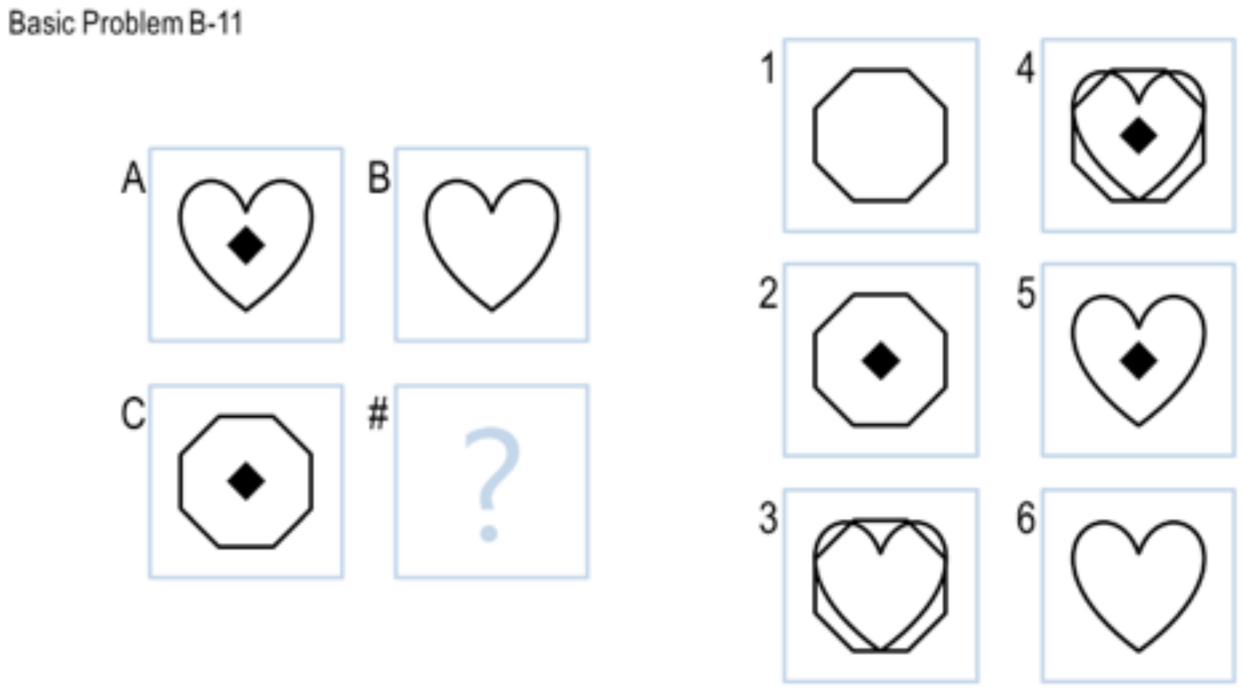
In this problem, an obvious observation is that all of the objects are circles. The challenge that arises to the generator, therefore, is which circle in **A** map to which circles in **B**. This mapping questions is not hard for a human to answer visually, but for my generator, extra steps are required.

Any time multiple objects are found in a figure, the generator first calculates the difference in number of objects between **A** and **B**. In this case, we notice that Figure **A** has 5

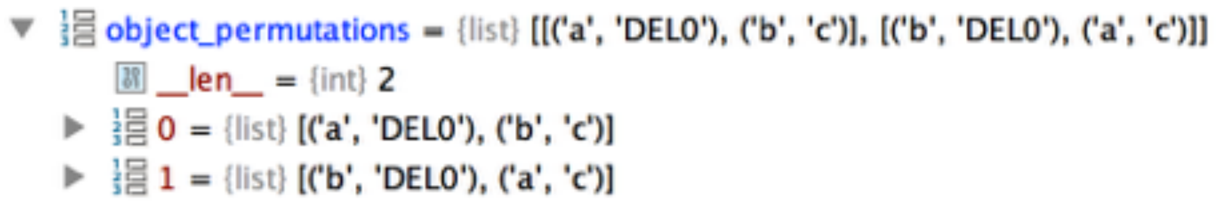
Figure	Objects	Difference
1	1	2
2	3	0
3	3	0
4	3	0
5	2	1
6	5	-2

objects, while Figure **B** has 3. Therefore, the difference between the number of shapes between **A** and **B** is **2**. Without further computation, the generator can already eliminate the answers from the answer set based on this simple fact: whatever transformation were to take place from **C**, the end result should differ by exactly 2 objects[4]. In this particular problem, every single answer is filtered out except from answer **1**. The following table illustrates how this filtering process takes place:

Therefore, without further computation, the correct answer has been filtered out the generator having to generate any transformations. In cases, however, where there does not exist such obvious differences in figures, the generator continues by generating all possible object mappings. Consider this example from problem



After the initial filtering by the generator, the agent reduced the possible answers to either **1** or **6**. At this point, the generator must continue execution and determine how Figure **A** transforms into Figure **B**, and apply that transformation to **C** in order to choose the correct answer. To start, however, the generator must determine which which object in Figure **A** maps to which object in Figure **B**. For the human, this is very trivial task: it is obvious that the heart in **A** maps to the heart in **B.** The generator, however, is unaware of the shapes that are involved. It makes no distinction between the heart and the diamond, therefore, the only option is to assume that both possibilities are equally probable. As such the generator generates a permutation of all possible combinations between the objects in **A**, and the objects in **B**:



```
▼ object_permutations = {list} [[('a', 'DELO'), ('b', 'c')], [('b', 'DELO'), ('a', 'c')]]
      __len__ = {int} 2
   ▶ 0 = {list} [('a', 'DELO'), ('b', 'c')]
   ▶ 1 = {list} [('b', 'DELO'), ('a', 'c')]
```

[4] When speaking of differences here, I mean in terms of existence (i.e whether the object exists or not) — The number of objects in Figure C should differ by the same amount of objects that Figure A differed from Figure B.

As observed above, there are two possible scenarios:

Object	Transformation
a	Deleted
b	C

OR

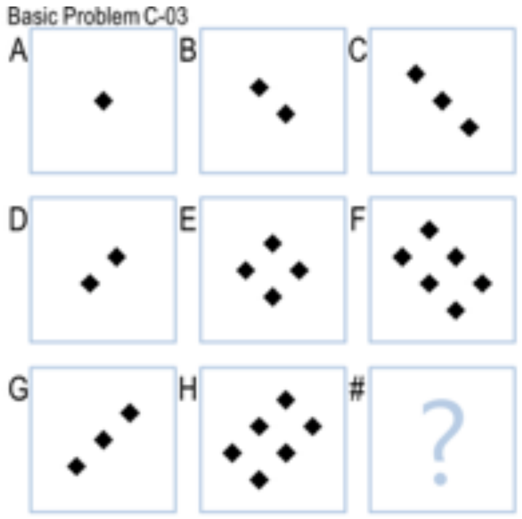
Object	Transformation
a	Deleted
b	C

Each possible scenario is observed by the generator, however, an important step that the generator performs, is it determines which transformation is quickest[5]. In this particular case, the quickest transformation From Figure **A** to Figure **B**, is to delete the *diamond* from Figure **A**. The generator, at this point applies this lowest transformation to Figure **C**, and passes the control over to the Agent Tester.

3x3 problems

For this project, in order to handle the more complex problem set, I made an attempt to solve the 3x3 problem set by creating a sub problem composed of the bottom-right quadrant[6]: E, F, H, #. My goal was to solve the 2x2 using the methodology I had already defined in project 1. This strategy, however, didn't workout as I had expected. Primary, my current alogrithm's use of object *diffs* to determine if two figures have the same number of objects[7], is incorrectly eliminating answers from the set which are correct. The cause of this seems to have been due to the fact that the 3x3 matrix has transformations in 3 dimensions (rows, columns, and diagonally). By solving only the 2x2 matrix, my algorithm was neglecting the surrounding relationships which were necessary to filtering out improbable answers.

I, therefore, introduced a diff function which generates a differ based on proportions rather than simply on object difference. This worked well for problems such as *Basic Problem C-03*, where the *diff* relationship between E and F is proportionally equivalent between the *diff* difference between *H* and *#*.



I critical fault which is clearly evident in the agent's performance is its failure to make necessary transformations between *H* and the candidate answers. Furthermore, the amount of permutations that the agent performs on these problems are amazingly high—crippling its ability to solve complex problems in a reasonable amount of time.

---

5 A "quick" transformation is one which requires the fewest amount of mutations.

6 This was inspired by James Allen (https://github.com/jamesmallen/gt-cs7637)

7 This metric is used to eliminate answers that are wrong because they don't have the correct number of objects.

Lessons learned from the 3x3 implementation

- The generator which generates the different permutations of transformations needs to be more intelligent as C-12 takes unreasonably long due to the amount of permutations that are possible (over 300,000!)
- When applying transformations to a figure, special attributes need to be considered. Categorizing the types of attributes involved would enable the agent to generate more accurate transformations.
- The knowledge base of the agent should be more unified[8], and working with that single source of KB.

3x3 Problems Visual Approach

It was only in this final project that I attempted a visual approach to solving the problems. Part the of challenge with the visual approach has been in getting accustomed to the Pillow library[9]. The feature that Pillow providers are not all necessarily obvious, thus, a somewhat significant amount of time was spent experimenting with simple image manipulation using the tool. Going to the visual approach from the verbal one, in fact, result in almost a compete rewrite of the AI agent. I had initially planned on reading in the problems visually, and generate the *problemText.txt* file, and then just solve the problem using the code that I had already written for the previous projects but, as I have soon learned, it was much easier to solve the problems visually using other means.

The way in which I was able to solve most of the problems in the given sets, is by using production rules. Going through individual problems, I tried to figure out my method of thinking when solving said problems. For example, a common pattern which I very commonly utilize when looking at a new problem is rows and columns similarities. That is, if simple features like number of shapes are increase across the first two rows, then there is a good chance that the number of objets would be increasing on the bottom row as well. Simple patterns like these were, in fact, very common in the problem sets. All of the rules which I defined for solving the problems maintain a similar *conditional* format:

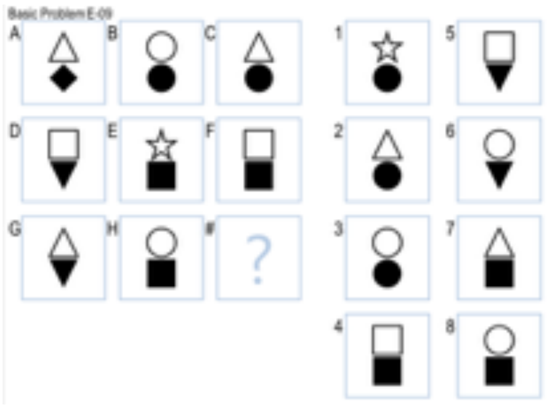
Rule #	Rule
2	**IF AxE = E AND BxF=F AND DxH=H THEN ExN=N**
8	**(A-B)+(B-A) = C AND (D-E)+(E-D) = F] THEN [((G-H) + (H-G) + F) + (DxF) = N**
…	…

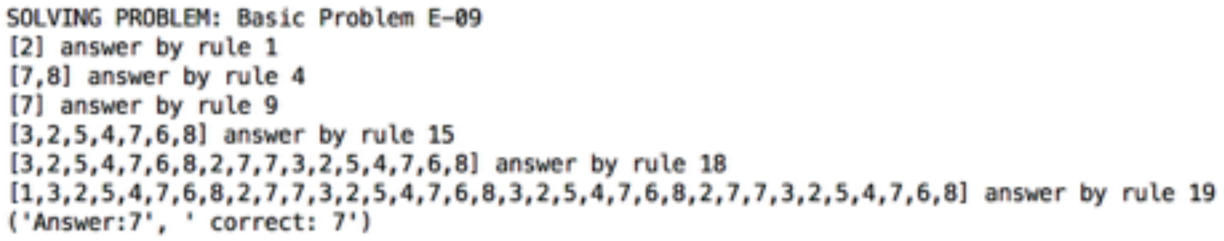
In total, I had 25 rules that each problem was processed through. Its important to note, that these rules are not simply definition to said problems, but they are in fact cascade in a more general sense to the overall intelligence of the AI agent.

[8] Currently, knowledge seems to be more scattered out throughout the app, causing redundant code to appear.

[9] https://python-pillow.github.io

For example, given **Basic Problem E-10**, 5 rules are actually being applied to it. Each rule is independent of the others, thus they each attempt to determine an answer (or answers) from the pool of answers for which the rule is satisfied. In this case, the following rules are satisfied in one way or another by the AI agent: [1,4,9,15,18, and 19]. Because rules cascade, each rule's pool of available answers is actually all the answers from the previous rule plus the answers from the rule before that one, all the way up to the original 8 answers that are given in the problem. A typical problem would yield a answer set as follows:





```
SOLVING PROBLEM: Basic Problem E-09
[2] answer by rule 1
[7,8] answer by rule 4
[7] answer by rule 9
[3,2,5,4,7,6,8] answer by rule 15
[3,2,5,4,7,6,8,2,7,7,3,2,5,4,7,6,8] answer by rule 18
[1,3,2,5,4,7,6,8,2,7,7,3,2,5,4,7,6,8,3,2,5,4,7,6,8,2,7,7,3,2,5,4,7,6,8] answer by rule 19
('Answer:7', ' correct: 7')
```

It is by design that some rules yield such a high number of possible of answers[10] — Each rule has a specific purpose. The more specific a rule is, the more useful it becomes in proposing valuable candidate answers.

In order to determine the correct answer from all of these possibilities, I developed a ranking system for each answer. It is a very straightforward way of getting the most plausible answer. Therefore, the highest ranked answer is, in fact, the answer which appears to be the most common from the answers proposed by the production rules.

Most of the rules that I utilized for solving the problem sets were *SET* operations (UNION, INTERSECTION, DIFFERENCE, etc). There are some rules, however, that take other types of transformations into consideration. For example, I used the flood fill mechanism to roughly determine how many shapes are in a particular figure. Furthermore, image rotations are used since some problems have rotated figures. Rations and flood fill mechanism were mostly implemented in the PILLOW library, as such, there was very little custom implementation on my part concerning these techniques. Lastly, I implemented a way of detecting whether two halves of a figure are moving across toward each other as in *Challenge Problem D-05*. This rule was created for problems as these by generating an image for each movement in a given frame, and checking it against the end frame[11].

Lessons learned from 3x3 Visual Implementation

As mentioned earlier, if a rule is too general it can be come counter productive. A general rule can produce false positives resulting in too many proposed answers to select from. It was important to constantly run the rules against the problems when ever there was an addition of a rule or modification. It has been the case that a rule would return many false positives which

[10] Rule 19, in this case is a rule that states "if A and E are different, N should also be different"
[11] This causes the agent to suffer a little from a performance perspective, however, it is not noticeably significant.

would critically cripple many of the other rules. It was necessary that I be aware of this as I was writing the different rules.

The shape detector which I implemented using the flood fill algorithm is accurate in most cases, but it becomes very inaccurate when small shapes appear in a figure. Small shapes typically result in a hanging 1 pixel artifact around objects which the flood fill algorithm would count as a separate shape. This caused a few problems for some of the rules, however, I was able to work around it by making sure that, even though the number of shapes is incorrect, the number should be consistent.

Lastly, I used a set of thresholds in the production rules. These thresholds mainly have to do with how objects are compared and constant values that are used in affine transformations. For example, I have found that in my case a similarity threshold of **40.7** was ideal for determining whether two figures are identical. It was somewhat difficult at first to determine a good similarity constant, but as more rules were added, this number became more and more clear.

Tester

Although the generator does much of the heavy lifting in this intelligent Agent implementation, The tester does, however, add another layer of intelligent to the whole program. For example, when a list of answers are generated, the ones which are not even part of the answer set are automatically discarded[12]. In the end, a list of candidate answers are left to choose from. For typical problems, the candidate answers list is usually very low. It is however, problematic when the tester is not capable of reducing the answer set all the way to 1 (i.e. the correct answer), or worse, reduce the problem set to 1 but to the wrong answer. A way to improve on the probability of arriving to the correct answer, I have learned that there can exist multiple transformations that result in a particular answer because of the nature in which the generator generates the transformations[13]. As such, I maintain a list of possible answers in a priority queue, and every time an transformation yields a particular answer, its priority increases. As a result, when selecting the best possible answer, the tester always selects the answer which has the highest priority in the answer queue.

For all of the given input problems in the Basic Set, the Tester was able to successfully select the correct answer from the priority queue data structure.

Mistakes and Challenges

As mentioned earlier, the Agent makes assumptions about the referenced object properties. For example, objects are not distinguished when they are “inside”, or “above” other objects. This could result in false positives when determining whether two figures are in reality identical. This problem is further complicated due the fact that all permutations of a transformation can exist in a given problem.

To remedy this issue, a data structure must be implemented to maintain reference mapping between these special object attributes. Implementing a mechanism to maintain the

[12] Note that at this stage in the program, The answer may be already reduced due to the workings of the generator prior to the tester taking control.
[13] Permutation of all possible combinations of object combinations.

references is challenging on many levels — for example, when considering property “inside: a,b”, the swapping of these references must also be considered, “inside: b,a”.

Other mistakes that my Agent makes are those that involve objects which have angles as properties. The agent is aware of how to interoperate angles[14], as such, it assumes a few select angles when encountering these properties. Furthermore, the Agent is completely unaware of shapes in figures. Because of this, the generator creates a list of all possible permutations. This can be very inefficient for figures which contain many objects[15].

Conclusion

Looking back, I think the Agent is fairly limited primary, however, by the way that the Agent is working with the information available. Some of the biggest challenges had been to determine how to process the *Verbal* data that problem provides. The Agent had to be made aware of this data structure and the special case properties mentioned earlier[16]. For better performance and generalization, its better to load the image directly using vision frameworks like OpenCV. This would enable the agent to read the image in a generic form agnostic toward the properties of objects in figures. Doing this, would certainly make the agent more efficient[17]. Lastly, a CV approach more closely mimics human strategies of solving these kinds of problems. For example, in CV the agent would not be concerned of how a particular object is labeled[18], but would just read that object in as data. Similarly, the human mind doesn’t care to have reference to an object, we just visually are able to reference it. Using the verbal representation of the problem data, my agent didn’t quite solve problems similar to how humans solve such problems due to its combinatorial nature and its trial and error approach. Furthermore, the agent does very little initial observations about a given problem. There is no concern of obvious patterns in the input image. Humans, however, have a distinct ability to look at a given problem and almost immediately be able to observe certain patterns and features which would otherwise go unnoticed by a KBAI Agent.

[14] A set of angles are hardcoded in order to detect reflection due to rotation of certain objects.
[15] Basic Problem B-12 has many shapes in a single figure. The generator generates 120 possible permutations of possible mappings between the objects in Figure **A** and the objects in Figure **B**.
[16] Reference properties like “inside”, “above”, etc
[17] There is a level of redundancy in how the agent currently calculates similarity. A CV approach for this would be a significant improvement.
[18] All objects in the Verbal representation data which was used by the agent had labels indicate how objects relate to each other.