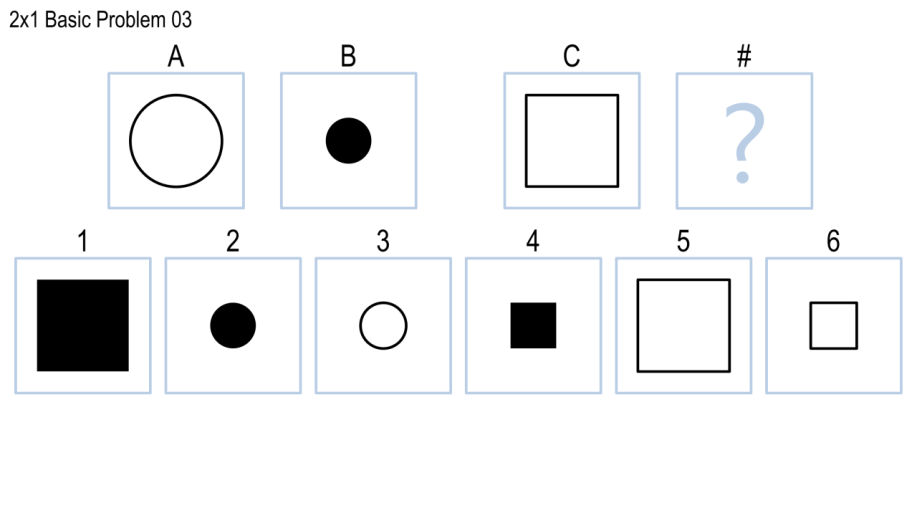
Elijah Philpotts

Project 4 Design Report

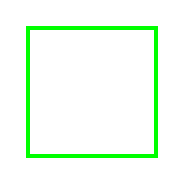
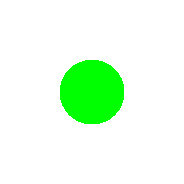
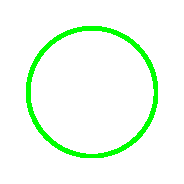
Introduction: Overview

As a side note, I have a feeling that as a whole, the Raven’s Progressive Matrix projects were very challenging in the beginning; but as you work on them and gain an understanding of proper representations and patterns to utilize, they become very manageable. If I wasn’t working and taking classes, I believe I could have made an agent that could have solved all of the practice problems with just a little more effort; but considering time for me was a big factor, it was not possible. This version of the project however was very challenging for me. I have worked with statistics; and am currently taking Machine Learning; so the mathematics of Computer Vision did not frighten me. With that being said, trying to learn something so complicated on the fly is not easy. The difficult part for me was not the reasoning part because I could have simply re-used the row-pattern method (which will be explained in a later part of the paper) from Project 3. However, getting OpenCV to recognize the objects in the first place was the trouble. When something went wrong, there was usually little documentation as to why it went wrong. So all in all, I believe if I had taken a computer vision course and was able to sit down and learn the ins and outs of why OpenCV works, I believe I could have made a better effort to solve the problems. But I do have a few ideas of how I could go about solving the problems in more unique ways that are a bit more complicated than my row pattern method; but may end up generalizing better than before.

Part 1 & 2: Reasoning and Correct Answers

First, I am going to start with the reasoning I used for my agent. I called it the Row-Pattern method; where we focus on what has changed between networks as a whole. It was created from the propositional 3x3 matrices for project 3 in an attempt to hopefully streamline the agent’s reasoning into something a bit more general rather than trying to pick apart smaller details within the objects. So for example, 2x1 Basic Problem 3 shown below:

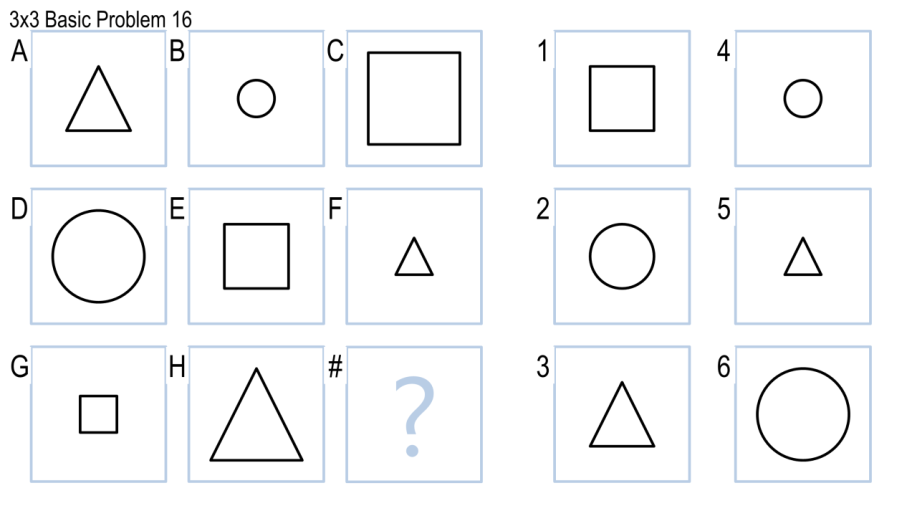
Let me first explain how my agent would handle visually obtaining the attributes from the shapes. Then, I will explain the actual reasoning once we have the attributes. For the above problem, I would first utilize a code segment found in the piazza forums to color each of the Networks’ shapes with colors. I choose to use primarily green and yellow to color the objects because darker colors made OpenCV’s findContour function have a hard time picking up edges and shape boundaries when converting them to grayscale. So for this more trivial example, the network would color Network A through C in the following way:



From here, the images would be converted to grayscale and then we would find the contours for the images. It then would classify the shape based on the number of contours it has (so for example, the first two would (hopefully; I’ll explain later) classified as circles; and the last one would be classified as a square. For the fill, I essentially for sort of lucky because I tried to check for the solidity of the object; but it always returned close to 1 for filled and un-filled objects. However, the filled objects always have 2 contours for a single object; while the filled ones have 1. So if one object had one contour, we would classify it as filled; while if it had more than one, it would be classified as no fill. To determine size, I simply just calculated the area and used that against thresholds for the shapes (if the area is below x, then it’s small; if it’s above y, then it’s large, etc.). The angle was a bust; which I will explain in the mistakes section; but these major elements are what make up my agent. I also wanted to include finding the locations of objects; but that also failed miserably despite the methods that I attempted. So this is an outline of the visual aspects used to get the attributes from the figures as opposed to propositional representations.

Now propositionally, it was much easier because I could just take the representations from the text file and then put them into RowLists; which contained information about what attributes changed in the rows. So for the above example, the RowList dictionary “Shape” key would have contained a 0 value because the shapes did not changed across the Rows (from A to B). However, the “Fill” key would have contained a 1 because the fill changed across the rows. Also, the “Size” key would have had a 2 because the shape got smaller. So we take this information into account for Networks A and B; and re-use it from Network C to 1-6. So in total, we have 2 RowLists: one for Networks A and B and another for C and the answer networks. We then compare what the RowLists look like to see if the changed match up. So if we had a RowList for Network C and answer 1, it would not match the RowList for A and B because although the “Fill” and “Shape” Dictionary values would match, the “Size” would fail because it would have a 1 instead of a 2 because the shapes didn’t change for C and 1. This worked fairly well; as I was able to get 15 out of 20 on the 3x3 problems using this method.

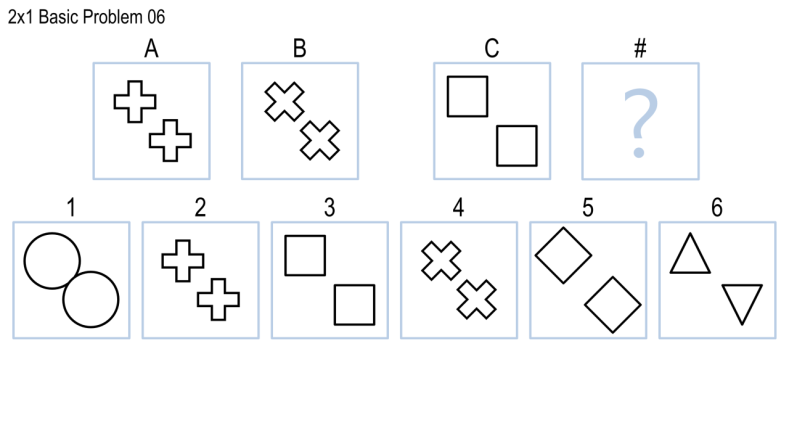
One more example to illustrate is shown with the problem below:



With a problem like this, we couldn’t simply check if the objects stayed the same or changed; which is where the RowPattern name comes from. What the agent would do here is realize that there wasn’t a consistent ordering to the shapes in the problem. However, it would pick up on the fact that there was a triangle, a circle, and a square. It would also pick up on the fact that there is a small, medium, and large object. So for the RowList, we would have a 3 for the “Shape” key (which means that the objects throughout the row are different) and the “Size” key would have had a 3 (which means there is one of each size of shape in the row). I made the agent this way because it would generalize better to future problems if we didn’t put emphasis on how the objects were different; but just know that there is a difference and that the differences will more than likely match the ones in the first row to some extent. So if we chose an answer network that had a repeat element, the RowList would catch it because it would not have a 3 as its value; it would more than likely have a -1 as a “no noticeable pattern”. So in this case, all of the answer networks would throw a false for a match except for the correct answer 2. Of course, this reasoning tends to fail when you have trouble getting the correct attributes from the visual representations (this will be elaborated upon in the mistakes section).

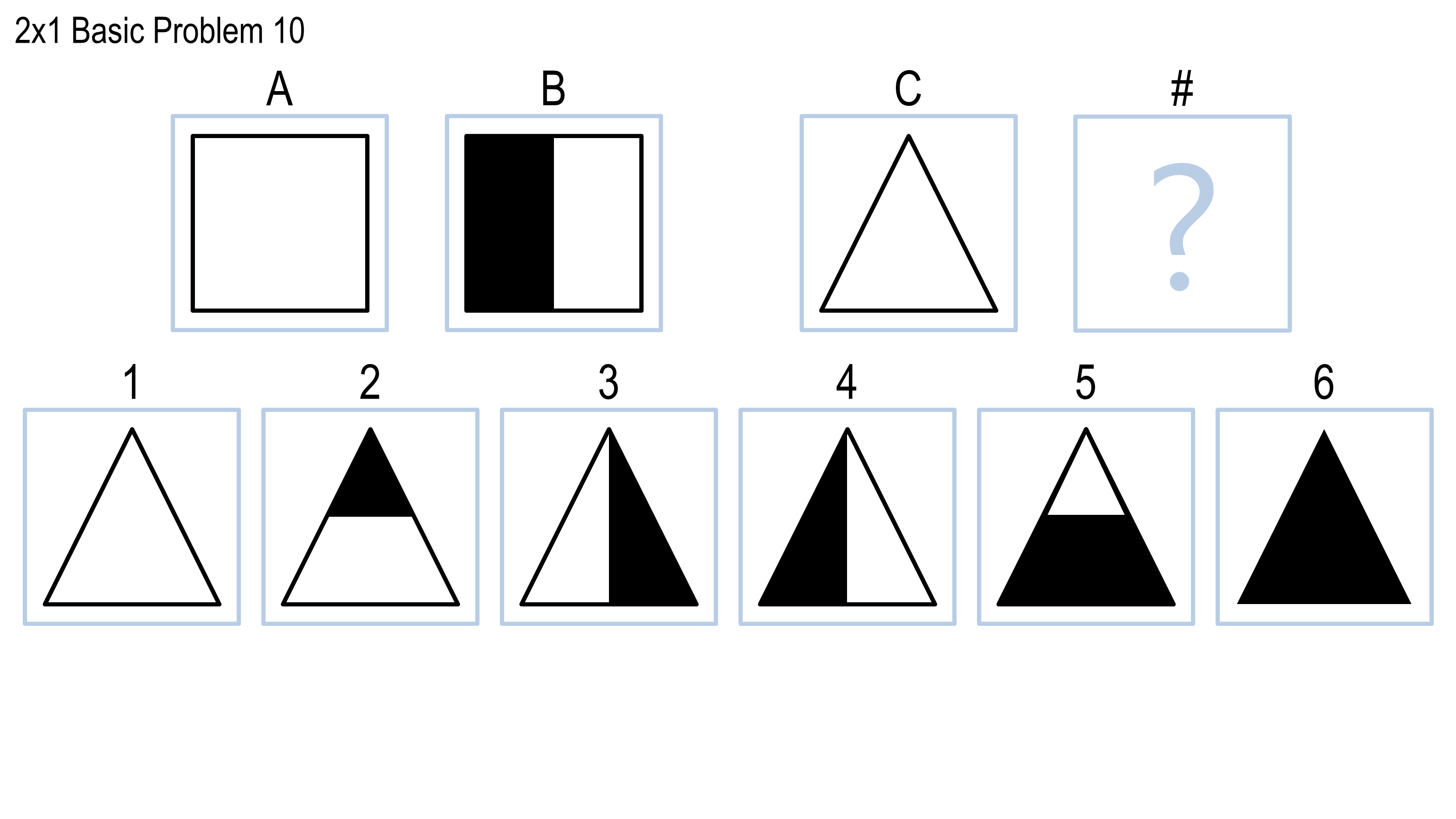
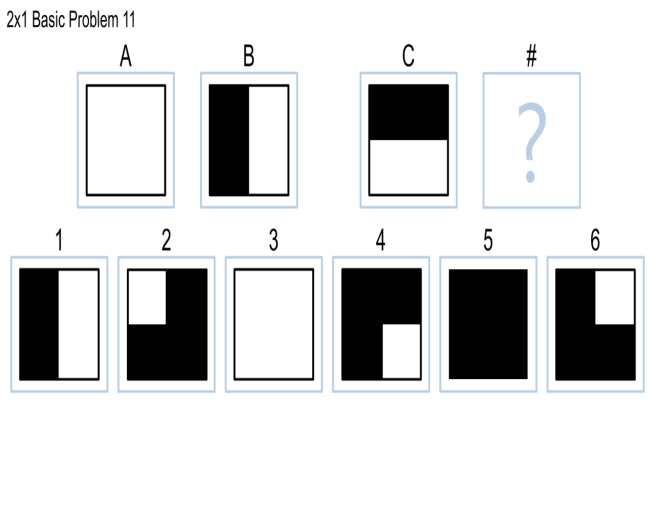
Part 3: Mistakes

My agent made quite a few mistakes due to the visual reasoning aspect of the project. As starting place, my agent had trouble getting consistent angles from the object. The problem below will illustrate this:

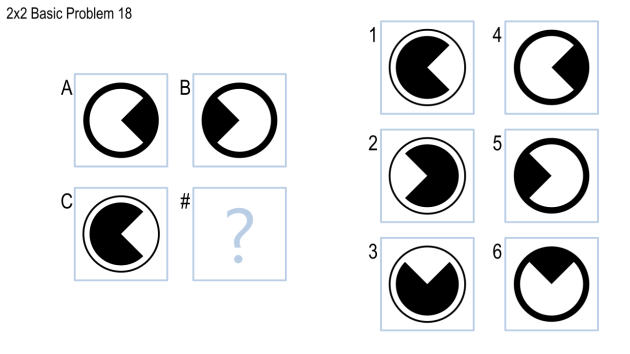
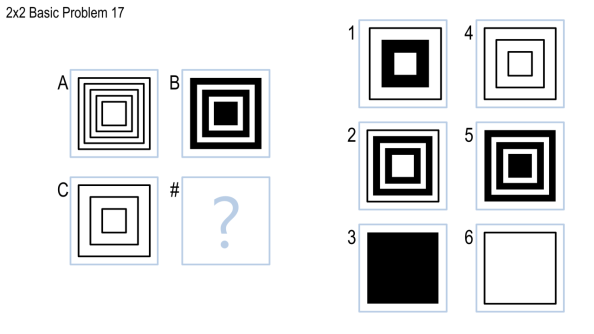


For the above problem, we know that the RowList would contain the fact that the “Shape”, “Fill”, and “Size” keys would contain a 1 because the attributes haven’t changed. However, we can see that the crosses have rotated. This normally would be easy with a propositional representation because we would just look at the angle and see how much they rotated. However, the visual representation isn’t always as precise; frequently, when trying to extract the angles, I would get angles that were off balance such as 270 degrees for one object in Network A and 225 for the other object in Network A. This causes problems because the angles in reality have stayed the same. This causes agent to fail because normally, the angles in the Network stay the same until you move to another network (in this example, Network B). This also causes the agent to fail because of another issue with the angles: with the squares, the angles would always return 0 because the objects did not have over 5 contours (findContours would throw an error). I tried using the moments to calculate the angle; but it would return the same angle even if the object rotated. So trying to find the angles for the squares from this problem was very difficult; causing it to get what would normally be a fairly easy problem incorrect. This also highlights a difference between propositional and visual representation because with the propositional representation, I would have been able to utilize my agent’s reasoning to immediately solve the problem. However, I now have to ensure that the visual representation is correct; otherwise, no matter how good the agent’s logic is (and I’m not saying mine is necessarily good), garbage in, garbage out.

Another area where my agent fails is trying to detect is partial filling as the following problems:

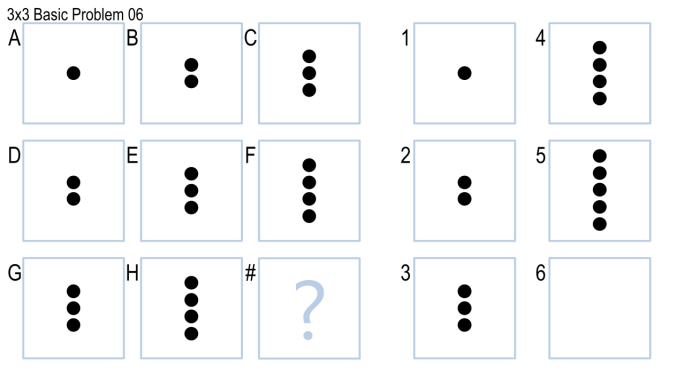
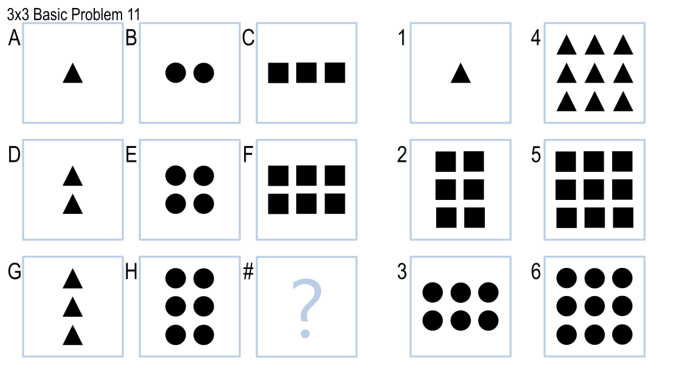
 

For problem 10, the propositional representation, we would just match up where the fill was with the location of the object. Now granted, my agent’s reasoning usually did not take location consideration; but here, it would be an easy patch to add some code that will react if we have a half fill on a certain side. However, for the visual reasoning, it is a bit more difficult on the detection side because we don’t have explicit information on where the fill is spoon-fed to us. If I use the findContours method, we will get two different objects returned; so that doesn’t do us much good. One way around this (which I didn’t figure out how to do) would be to overlay the older object on the new one and detect how much of the fill is there and where the fill it. However, this method is problematic because the findcontours thinks that these are different shapes. This makes things much more difficult because we now can’t overlay the shapes because we’re dealing with two different shapes. The same thing happens with Problem 11: my agent can’t properly detect where the fill occurs; so it cannot pick a correct answer. However, we have to expand on this because unlike problem 10, a simply fill matching wouldn’t work because of the nature of the problem. We would have to detect that we are only looking for fill on the left hand side; so this is an added dilemma to the problem (I will present a potential solution in the next section).



The second-to-last area my agent failed in recognizing objects inside of other objects (especially if they were filled). I did not have any method of determining if an object was inside another object. This was not in my agent’s reasoning; nor was it in my visual detection system. The reason why it was not in my agent’s reasoning was because I could not find a suitable pattern that would generalize well to other patterns. This was due to the mass number of locations the propositional representation threw at my agent. The primary issue I had was that most of the problems could be solved without taking location into consideration. Taking them into consideration actually caused some of the problems to come up with incorrect answers; so this is a shortcoming of the RowPattern method. In addition, detecting Pac-Man using the contours was fairly difficult. I found that it usually had 13 contours; but then again, so did some of the smaller circles. So we would misclassify examples if we took Pac-Man into consideration. I also tried a bounded circle method; but it would read in each Pac-Man with a different return value. So the inconsistency led me to abandon this method.

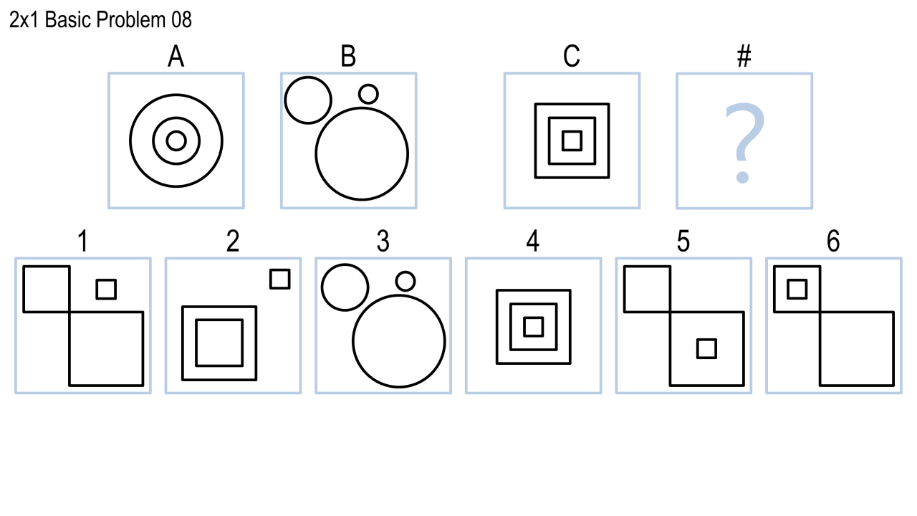
The last area where my agent failed was not because of faulty reasoning; but because I had trouble consistently detecting the number of objects that were in the object. Problems such as the following gave my agent trouble:



Sometimes, it would detect the correct number of objects. But others, it would either detect an incorrect shape (it liked to classify smaller triangles as half-arrows and squares) or would not detect all of the objects. So for example, for Network F in problem 11, it would return 5 squares as opposed to 6. In network E for problem 6, it would return 3 circles and one triangle. These types of errors are what caused my network to fail on problems with additional objects.

Additional Part: Complex Methods to Simulate Intelligence from your Agent

David spoke about a potential method being used for the project and to highlight how it would be used here. Well, I am going to do something a bit different: I’m going to highlight various methods that could be used and highlight some problems they could be used for; as well as some pesudocode to accompany them. I wanted to go all out for this last project and incorporate as much as I could into the reasoning by utilizing the subjects we learned in class because I believe contrary to popular opinion that many of them can be used for the project. I’ll be honest and say that I didn’t implement these myself because I had the idea; but could not research an efficient way to implement the solutions. I hope that this design report may give others in future classes some ideas about how to approach the problems; in the hopes that they can succeed where I failed. In addition, I believe that a good agent will have multiple methods of solving problems rather than one because there is almost no way one single method can generalize to all of the different problems an agent may face; let alone simulate human cognition. My RowPattern may get the job done fairly well; but a true AI agent will have multiple sources of reasoning to go with their primary method. All of the problems listed below are problems I thought were interesting or ones where my agent failed; and my original plans to mitigate this failure. To describe the some of the reasoning techniques I would like to have imparted to my agent, let’s look at problem 8 in the 2x1 example:



As humans, we would notice that the circles are inside of each other; with the circles decreasing in size in a consistent manner. We may even draw comparisons to other objects that are similar to the objects presented in the network (for example, Network A looks like a target). This could be implemented in an agent via Learning by Recording Cases. We do not want to use Case-Based Reasoning because we don’t want to alter the cases to make a new case. We just want to see how close the original one is to the Network we’re currently analyzing. This would not be necessary to solve the problem; but would help the agent simulate human cognition. It could be implemented in pseudocode as follows:

Program LBRC.py

networkShapes = identifyShapes(Network)

array\_of\_patterns = identifyPatterns(networkShapes)

//array\_of\_patterns could be attributes such as: 3 shapes, all //circles, Circles are each inside one another with the //smallest, etc.

For Case in Cases:

If kNearestNeighbors(Case, Network) < ε:

currentSimilarCases.append(Case)

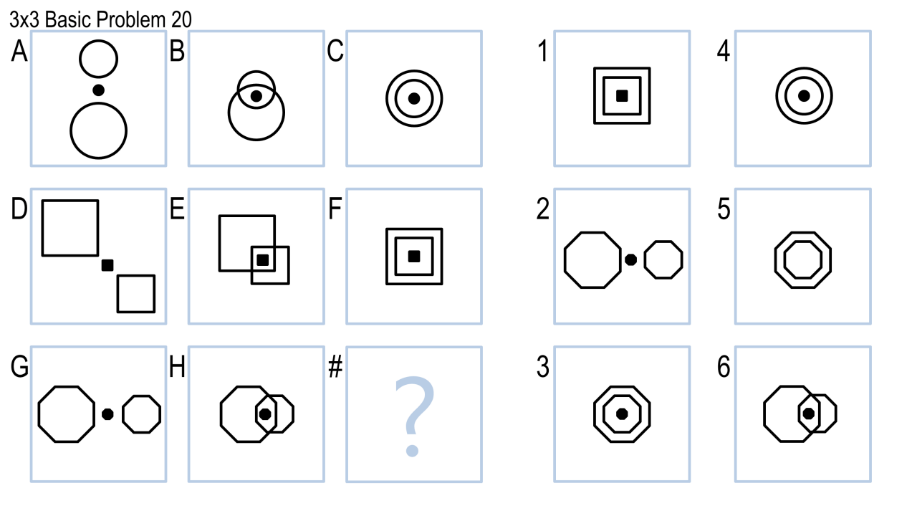
Return currentSimilarCases

Distance measures for K Nearest neighbors could be attributes such as how close the shape locations in the Network match up to the ones in the Case, how many shapes are in the Case compared to the Network, etc. To obtain array\_of\_patterns, we could have taken by the visual reasoning route and used a distance measure for how close the similar pixels in the image are to each other; or we could have just taken a more propositional representation and looked at patterns in the shapes that matched up. The second method would be less computationally intensive; but the first one could be tweaked with a bit more precision because number of pixels is probably a lot higher than the number of attributes.

Both approaches would however require a vast amount of not only processing power; but memory overhead as well because we would have to keep so many cases. However, the method above would be a step in the correct direction for simulating human cognition and is something I would like to have pursued to add to my agent’s reasoning.

Then, we look at the Network B and see that there are a bunch of circles of varying sizes that are now scattered. As humans, we would probably first hypothesize that the circles are simply scattered now rather than stacked inside of each other. We would then look at Network C and see that we have squares stacked on top of each other. We then would probably proceed to pick 1 (which is the correct answer) because all of the answer choices are either the incorrect shape in Network C (which we hopefully would have caught didn’t change for the first two networks) or have objects inside them. Now, we could use LBRC.py to find out what objects are similar to the objects in Network C; but I do not think this would represent true human cognition. The reason for this is because most people would recognize network A as a target; while a group of squares really wouldn’t bring up anything familiar as quickly. So now the agent just got more complicated because now it would have to figure out what objects it should try to find similar cases for and which ones it shouldn’t. A time constraint would be one way of solving this problem; so that if it doesn’t find a suitable match after a certain time period, then we just skip that step and begin to solve the problem using basic reasoning methods. So for this problem, it wasn’t the actual reasoning and solving the problem that was the difficult part; it was walking through the steps of simulating human cognition.

A Problem that builds onto the last one would be 3x3 Basic Problem 20:



Here, we have the opposite situation in that this time, the objects are scattered; but then they come together. When using human cognition, most humans would glance at the problem and deduce that you only need the first and last networks to solve the problems. We would see that the objects were at first separated; but then are together. We would also see that the shapes have not changed; and that the middle one is filled. Using this methodology, we would deduce that Network 3 is the most appropriate answer. Now how does this problem relate to the last one? Well, we once again have the idea that most humans would deduce that the object in Network C is a target. We could also deduce that Network F is not a target; but looks very similar to a target. This would not have helped us much in the last problem; but may help a bit more in this one. Using our LBRC.py algorithm, we could actually plug in Networks 1-6 (or the answer networks) and see which ones most closely resemble a target. I believe this would simulate human cognition a bit more than comparing the exact placement of the shapes for this particular problem because we have already essentially computed the placement of the shapes in the Recorded Case with the target. So our next step would be to see which answer choices most closely resemble a target. We may want to relax ε a bit for this particular case… But wait, how do we as humans know when to relax certain assumptions about a target? How can we endow our agent with this ability to realize when to relax assumptions? Well, one way would be to do it would be the following:

similarCasesTieBreaker.py

for in Network1.Attributes && Network2.Attributes:

// Attributes is an array that contains attributes such as // shape\_placement, fill, how close the shapes are, etc,

if shape\_placement in Network1 and Network2 are within 𝛿 of each other:

Weight += .30

if shape\_fill in Network1 and Network2 are within 𝛿 of each other:

Weight += .25

if shape\_distance in Network1 and Network2 are within 𝛿 of each other:

Weight += .25

// …… something along those lines with all attributes

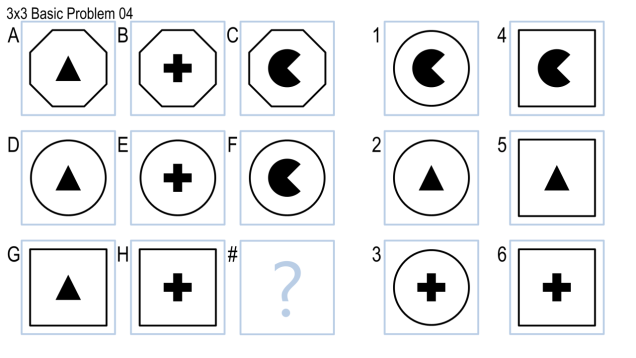
bool Execute = Weight > someThreshold

if Execute is True:

Execute LBRC.py with a decreased ε

This is similar to the way some people did their projects where they used a weighted approach and picked the Network with the highest weight as the correct answer. Here, we are taking more of a single perceptron neural network approach in that we are seeing if the Weight is greater than some threshold to fire and execute LBRC.py with a decreased ε. However, this method does still carry the heavy burden of having a large memory and processing overhead because we have to not only hold all of the cases; but iterate through all of the cases in order to find a suitable object that represents the objects in the network. So we have determined how to find out if ε needs to be relaxed; but by how much? If we’re too strict, then our objects may not find a suitable match in the cases. But if it’s too flexible, then we may have multiple objects that are returned; which will cause us to have to solve another tiebreaker. A way to decide how much to change ε would be to use the weight: if the weight is close to the threshold, then we may want to adjust ε down because it means that we’re close to a match; but not as close as we would like to be and vice-versa if the weight is higher. We would combine this method with other basic reasoning methods such as seeing that the shapes, fill, or angles change or stay the same in the Network. I thought this problem was good to place right after the last one because it built on the one what we already had with trying to compare shapes in our case list with ones in the network of objects.

I would like to give one last example that may be a bit more nostalgic than the last two. Let’s look at 3x3, Problem 4.



Here, we can see that all of the outer objects do not change; but the inner ones do. The RowPattern method could easily distinguish these features (assuming that we were able to visually pick them up) and pick the correct answer. However, how would we as humans approach this problem? We would probably see the last answer as missing a shape; specifically a Pac-Man. This is very important because most agents would simply notice the shape is missing without actually making the connection between the shape and something that we know. So we could our Recorded Cases similar to the target to find out that the shape looks like Pac-Man. Sure, this is a fairly inefficient way of solving the problem; but most individuals would look at the last row and notice that a Pac-Man is missing. They would then proceed to find Pac-Man with a square around him/her as opposed to finding just a missing shape.

One last point on the above problem would be if we changed Pac-Man to have a bow at the top of this head. Well, we may not have a case for this; but we know that it’s similar to Pac-Man. So we can use Case Based Reasoning to create a new case for this. Some pseudocode could be used as following:

Case-Based-Reasoning.py

for cases in Cases // these are the cases from Case-Based Reasoning

if case is within 𝛿 of our Pac-Man object:

create a new case called Ms. Pac-Man

Map the relation to Pac-Men

return the case that we found

For the comparison method in the if statement, we could do a pixel check and if there is an exact match, we know we’re probably just looking at another Pac-Man (which we already have). So we want one that is at least 𝛿 deviations away from our current Pac-Man Case. Visually speaking, we could do a pixel comparison to see how well they matched the object. So we now have what our new case looks like; hw do we add this case to our cases? We could then use Explanation-Based Learning to seal the deal to get the actual information about the case. An example would be as follows:

Explanation\_Based\_Learning.py

Old Case = Pac-Man

New Case = Ms.Pac-Man

Distinguishing\_Feature = featureExtraction(Ms.Pac-Man) –featureExtraction(Pac-Man)// What featureExtraction does is take the // features from Ms.Mac-Man and subtract them from Pac-Man. This would // be done on a pixel basis; so we would be left with a Bow.

New Case = findCase(Distinguishing\_Feature)

// So what the above function does is find out what exactly the // distinguishing feature is. We know that it’s a box; but does the // agent? Think about it as having our agent look it up on google ☺.

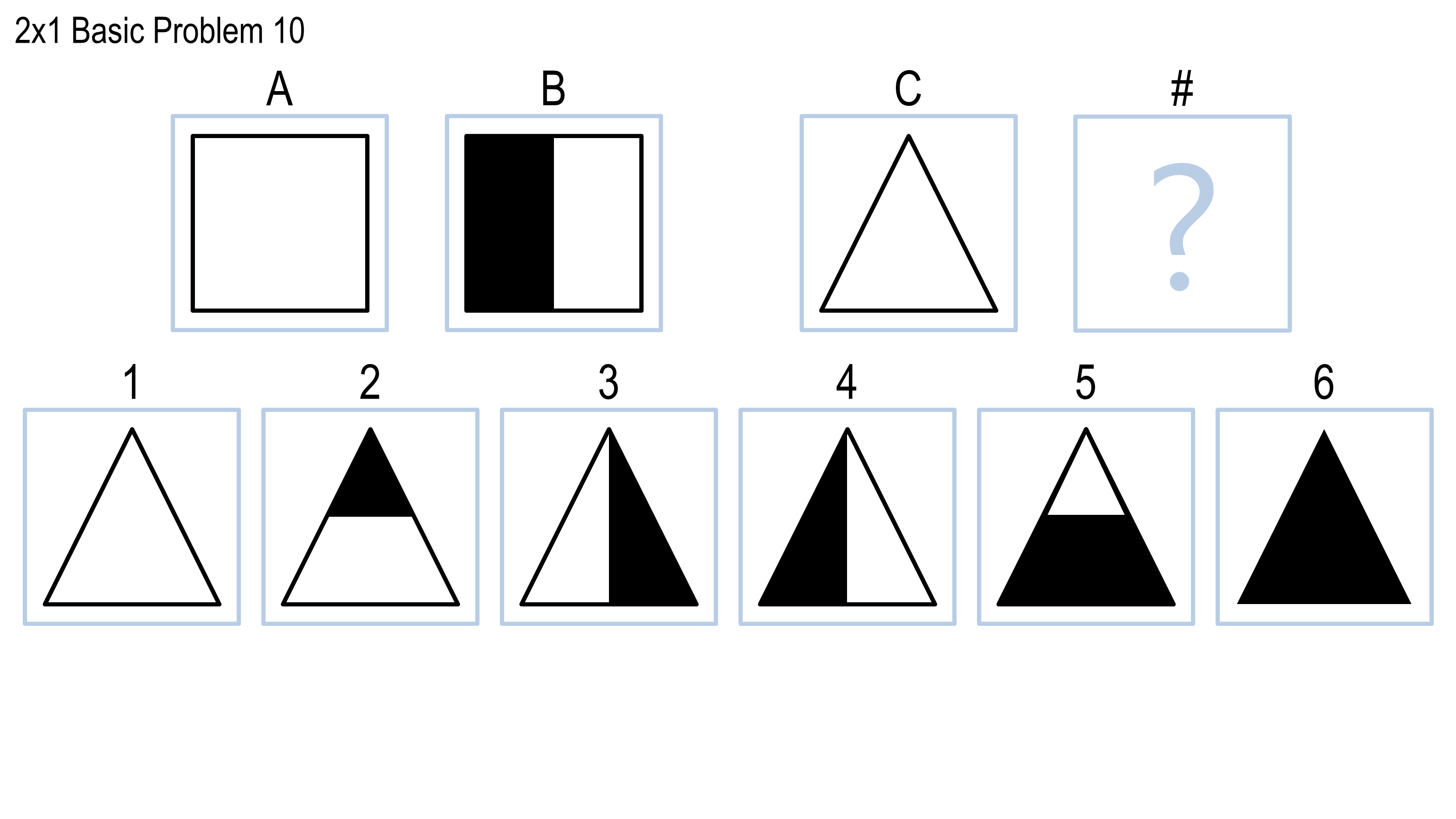
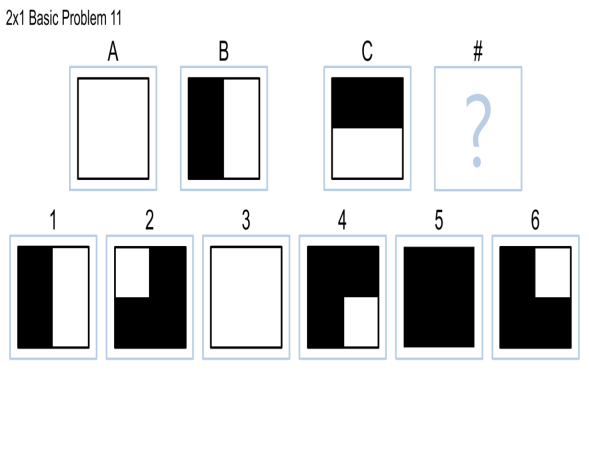
// After our agent finds out what the object is, we then move to the // EBL part of the algorithm; where we add a new rule to our agent:

Ms.Pac-Man = Wearing(Pac-Man,Bow)

AddCase(Ms.Pac-Man)

Here, I would definitely say that visual reasoning beats out propositional reasoning because when you have multiple objects inside one another, there can be a small memory overhead of having the correct locations such a left-of, right-of, etc. and then having a function that can accurately compare these. Visually, it would probably be much easier to see if an object is inside of another by doing some pixel analysis (something I could not get to work for the life of me). So visual analysis would take the cake with problems where location is present in the answer.

Here are two good examples of problems that we might approach using MetaReasoning:



This one took me a second to fully grasp what was going on with problem 11. At first glance, we might just see that we added fill to the object. So logically speaking, since there was a 50% fill added to the first object, there would be a 50% fill added to the next object too. Well, that’s incorrect; and trying to endow my agent with the ability to discern problems like this was a challenge for me (one I was not able to overcome). So how do we know what to do here? We’ve already established that we cannot simply check the fill; so we have to look at the location of the fill rather than the actual fact that it’s filled. I believe that this would be a good place to use metareasoning. A basic agent would hopefully realize that it got the problem wrong by using the basic fill method; so after that, it would have to think about what it did wrong. There are two ways to approach this problem: we can have the agent walk through its logic on how it solves the problem; and then find another approach by using Recorded Cases or possibly Case-Based Reasoning. Or we could simply have that the agent realize that it made a mistake and then attempt to find an answer to the question by using another method. Both methods are valid with respect to human cognition because some humans may just see that their method didn’t work and attempt to find another one. But others may want to find out why their thinking failed so they can add it to their Recorded Cases so that the failure has a lower probability of occurring again. Let’s assume that the agent wants to walk through its logic again; how would we implement that in code? We would probably start with having the agent find out what is used to determine the answer. So let’s say for this example that because there aren’t a lot of attributes with the shapes that we use the following code after we have been given the correct answer (note that this method can be used for visual reasoning; the algorithm may have to be tweaked for propositional reasoning):

MetaReasoning.py

// We first pull the answer choice into a variable

CorrectAnswer = getCorrectAnswer()

// We then show the attributes that stayed the same and the ones that // changed from Network A to Network B.

SameAttributesAB = {‘Shapes’: “Square”}

ChangeAttributesAB = {‘Fill”: 50}

// So for the above two networks, the shapes stayed the same; and the // fill increased by 50%.

// We now want to compute the above dictionaries for the Network C and // the answer Network. So our answer choice’s AttributeList is below:

SameAttributesCD = {‘Shapes’: Square”}

ChangeAttributesCD = {‘Fill”: 50}

// Our answer would have relied on AttributeListsAB to arrive at the // above AttributeLists for C and D and would have the picked answer // choice 5 because that most closely resembles the change in the // AttributeListsAB. However, we know this is incorrect. The correct // one is shown below:

SameAttributesCD = {‘Shapes’: Square”}

ChangeAttributesCD = {‘Fill”: 25}

So we now have the actual answer AttributeLists so that we can begin to determine where we went wrong. So a preliminary analysis suggests that we should probably see what differences our answer had with the correct answer. A quick comparison of the two suggests that we were off by 25% with our fill and that the shapes have not changed. So our focus should be on the fill rather than on the shapes this time around. Now for the Metareasoning part: we’ve identified that we messed up and identified what part of the problem potentially caused us to mess up. Now, my method revolves around answering certain questions about the problem to see where we went wrong. This is because if we try to simply use the fact that the new answer has a 25% change in fill, there are 3 possible answer choices that fit this criteria. So an examples would be the following:

MetaReasoning.py (continued)

// QuestionList is an array of functions that will be executed to // determine if we can retrieve the correct answer by utilizing them. // This may not simulate true Metareasoning because we aren’t // essentially coming up with a new way to think by thinking about // where we failed; but we are thinking about our thought process by // acknowledging that it needs to be changed.

QuestionList = []

For questions in QuestionList:

// the question we want to answer here is “where is the fill // located”? After we see this, we go back and utilize that // information to determine the location of the fill in the // answer choices.

for network in TotalNetwork: // we have a total of 6 answers; so // 6 networks

result = question(NetworkC,network)

if result == realAnswer: // we’ve found the correct answer

break

Notice how this can be used with a propositional and visual representation; but would probably work best with a visual representation in this case because it could simply reason that if we add the fill from Network B to Network C, we can simply use a simpler reasoning method that attempts to match up the fills from the problem. We also could bring in Learning By Recording Cases and add this problem to a list of problems that cannot be solved through conventional methods to be on the lookout for similar problems in the future. If we run across a problem that’s similar, we use the function that we used previously to solve the new problem. One way this algorithm could be changed is that we could allow the question loop to continue to see what other functions may return a correct answer. We could then take it a step further and see which one takes the shortest amount of time and tell the algorithm to make that one the function of choice to add to our cases when dealing with a problem similar to that one.

When comparing propositional and visual representations with Metasreasoning, a combination of the two would probably work best because some problems may work better with propositional representations (for example, problems with a lot of shape changes would probably do better not having to constantly compare pixels) and other may work better with visual representations (the one above). However, the agent would have to have some sort of criteria to decide whether to use a propositional or visual representation; maybe a function that looks at certain attributes of the problem and determines based on weights what to use. Or maybe even Recorded Cases of problem times for various problems; and based on attributes of the problem, we decide which one is most similar and then use the representation that we used for the selected problem.

Part 5: Time Factor

My agent takes around 3-4 minutes to run all 60 problems. Overall, I don’t think it’s too bad because there are a lot of inefficiencies I wish I could have patched up; such as not saving the files each time. My goal here was to get it to work; then work out the kinks and improve everything. The problems that will take the longest are the 3x3 problems and the ones with answers at the end of the networks. The RowList method doesn’t really make room for tiny improvements because it takes a higher level overview of the problems. This makes it more difficult to make improvements to the system because if you do, it has to check each problem for these improvements. It’s hard to detect them before they occur; especially if you are doing a visual analysis. This goes back to exploration vs exploitation: with propositional, we know what information we have and don’t have to look for additional information. With visual representations, we have to look for these patterns because we don’t know for sure that they don’t exist. Some can be easily found such as the half fill; but others such as 3x3 problem 18 with the filled squares can be more difficult to instantly deduce immediately.

Part 6: Human Cognition

My RowPattern method caters to the fact that most humans do not notice small details at first glance to solve a problem. This is because most of the time, we do not need them in order to arrive at the correct answer. So it would be a waste of time in the long run to look at the exact objects and their details. This is where the RowPattern method comes in: we examine the overall theme of each network and get a high level grasp of the objects; then we go into detail if necessary. Many problems can be solved this way; however, as I have seen with my agent, more detailed methods are needed if my agent fails at the general method. I believe my agent is in an perfect position to expand if I was given the proper time to do so. I have a general outline for the reasoning; now I need the next step (which could be some of the methods I outlined above); which is more specific methods that we as humans use to determine patterns that general methods simply won’t find.

Part 7: Visual vs. Propositional

Because I’ve probably beaten you down with the design report by now, I’ll make this section short and to the point. The propositional representations were a lot harder to initially set up with regards to actually getting to attributes to load up properly; whereas with the visual representation, I didn’t have much of an issue with this because once you use machine vision to interpret the object, you just put it into whatever data structure you need to use. However, once you loaded the attributes up into the data structure, operations on applying the agent’s logic (such as the RowPattern method) was similar no matter if you used a propositional or a visual (I literally just dropped my code in from project 3 into project 4 for the agent’s logic). However, a big difference between propositional and visual (and probably the biggest difference) is that with propositional, you know beyond a shadow of a doubt that the information you receive is accurate and correct. When you load up the attributes, you know that they are correct and that the machine vision aspect didn’t just make a mistake. With the visual representation, the computer vision aspect of the project (at least for me) may have mis-interpreted a figure, which would cause the whole problem to fail. So visually approach the problem, we were fighting on two fronts: get the correct visualization information of the object and make an agent that could reason about these aspects correctly. However, the visual aspect is a bit more flexible in find out for example, that as object is inside another object and formations such as those. I gave an example where we tried to find out what objects the network looked like; and if it looked like a target, we would find other visualizations that still held the RowPattern attributes. But we would also see if it upheld the fact that it looked like a target. Methods such as these aren’t as available when using propositional representations. So in summary, propositional representations seem to represent a trade-off between exploitation and exploration (similar to machine learning). With propositional representations, we can exploit the information that we have, even if it is fairly limited and doesn’t allow as to extrapolate other pieces of information like finding out if an object looks like a target, because we know that it’s true. However, with visual representations, it may take some fine tuning to ensure that each figure is read correctly and the correct attributes are extrapolated. Once this is fine-tuned, it opens up a whole new world of possibilities of incorporating various strategies into an agent’s reasoning.

Final Part: Conclusion

In summary, I listed out a bunch of methods that could correct some of the problems that my methods had because I felt like trying just one method would not help create an agent that can solve a multitude of problems while simulating human cognition. As humans, we don’t use just one method to solve our problems; so why should our agents? What I wanted out of this was to find a first line of defense so to speak. In the long-run, our agents should not only have a bunch of methods up their sleeves; but they should also have the proper reasoning to know when to use those methods given certain stimuli. So all in all, I have only scratched the surface of the RPM test because I have only endowed my agent with a general method. There are still many other methods that we can try; and the scary part about it is that the possibilities are almost limitless given the diversity of human cognition and the way people process and apply information.