CS 4635 - Knowledge Based AI

Brad Ware

Project 3 Design Report

Contents

1. How does it work?

I took the approach that my agent had last time and tried to build on that foundation. In Project 2 I implemented an agent that tried to find the most similar differences between figures and then returned the one with the best result. This time, because there are more relationships between figures than the 2x2 and 2x1 problems, I had to redesign my strategy and data structures to solve 3x3 Raven's matrices problems.

A. Strategy

From a very high-level perspective, the strategy is to analyze the differences between the relationship of figures ABC, DEF, ADG, and BEH. From there, document these mappings in wrapper objects which scores the differences between each of these figures. Then compare these scorings to the same wrapper objects that contain GHAnsChoice and CFAnsChoice. However, when comparing the scores, only the horizontal relationships were compared to each other, and vice versa with the vertical objects. Finally after scoring every answer choice, the one that returned the highest score was found to the the correct figure for the problem. If there were ties then there was another series of scoring for each of the figures that tied for the highest score in the previous round, eliminating the rest of the answer choices.

The agent stores figures ABC, DEF, ADG, and BEH in separate wrapper objects that gives a score to each group based on its differences in many categories. The same is applied to figures GHAnsChoice and CFAnsChoice. It then stores each group index in a hash table that indexes to another hash table storing the attribute category and transformation that occurred.

This same step is repeated for each answer choice and is then stored in a separate wrapper object with groups CF and GH.

After that, comparisons are made between the hash maps of CF and an answer choice, and the one's of groups ADG and BEH. This is also repeated for GH and an answer choice compared back to the hash maps of groups ABC and DEF. If the score is the same for an object, then the answer choice's score is incremented. Each level of similarity increases the score more, that way clearly wrong answer choices are eventually eliminated much faster. The figure with the highest score is then returned as the answer.

However, because ties can occur frequently, a list is used to keep track of all the answers that tied with the largest score.

If the list is greater than 1, the real elimination heuristics occur that compare many categories trying to find differences that can differentiate answer choices from another.

Finally, the answer choice with the highest score based on its similarity differential rating back to the Groups of figures (A,B, C), (D,E,F), (A,D,G), and (B,E,H) is returned as the answer.

B. Architecture

Four helper classes were defined so that one could easily compare and contain the information about each figure and it's relation to another.

Class Group (Ravens Figure, Ravens Figure, Ravens Figure)

The purpose of this class was to track the relationship differences between each of the objects in the 3 different figures. This class kept a hash map which stored numerical indexes for each Object and that index hashed to an attribute in a different table. These attributes stored a difference string trying to describe what transformation occurred from figure to figure.

Variables

- *fig1*: the first RavensFigure to be compared.
- *fig2*: the second RavensFigure to be compared.
- fig3: the third RavensFigure to be compared.
- *score*: the score of the relationship between the two figures.
- *scoreMap*: a hash map containing each object index as the key and another hash map storing the associated transformation with the attribute.
- *diffNumObjects:* string value that compares the different number of Objects that each figure contains and returns the resulted transformation.

Key Functions

- *setUpScoreMap: initializes the hash map with the correct object index and builds the helper map that goes into more detail about each attribute change.*
- *getfDiffNumObjects: returns the transformation from the different number of objects.*
- *getfFillDifference: returns the transformation from the fill difference of objects.*
- *getSizeDifference: returns the transformation from the size difference of objects.*

Class ObjectGroup (Ravens Object, Ravens Object, Ravens Object)

The purpose of this class was to track the relationship differences between each of the three objects in a figure. This class kept a hash map of characteristics and a **string transformation** keeping track of the differences. Any different characteristic was stored in the hash map, and it was taken into consideration the number of characteristics and the differing possible values.

Variables

- *obj1:* the first object to be compared.
- obj2: the second object to be compared.
- obj3: the third object to be compared.
- *attrMap1*: a hash map containing all of the attributes of obj1.
- *attrMap2*: a hash map containing all of the attributes of obj2.
- attrMap3: a hash map containing all of the attributes of obj3.
- *diffMap*: a hash map containing all of the associated differences in attributes of the three objects.
- diffList: a list of the attribute names that are different from each other.

Key Functions

- *compare:* made an ArrayList displaying all of the attribute categories that were different between the objects.
- *setUpDiffMap*: used the difference list to build a HashMap storing the category and transformation.
- angleDiff: used to track the numerical differences between angles.
- *sameShape*: determined if the object contained the same shape and stored that boolean value in the hash map.
- *sameSize/Fill*: determined if the object contained the same size/fill and stored that boolean value in the hash map.
- *getSame…* : getters for each of the boolean attributes comparisons
- *getValues*: getter for the hash map containing all of the attributes and their boolean values

Class AttributeGroup (Ravens Attribute, Ravens Attribute, Ravens Attribute)

The purpose of this class was to track the relationship differences between each Ravens Attribute in a figure. Simply stored the transformation difference between the two attributes that were of the same category.

Variables

attr1: the first attribute to be compared.
attr2: the second attribute to be compared.
index: the attribute's name that was used to store the category.
diffValue: a string that stored the difference between each attribute.

Key Functions

getDiffValue: returned the transformation between each attribute's value.
getIndex: returned the index/name that was categorizing this attribute.

Class Problem (Ravens Problem, Ravens Figure)

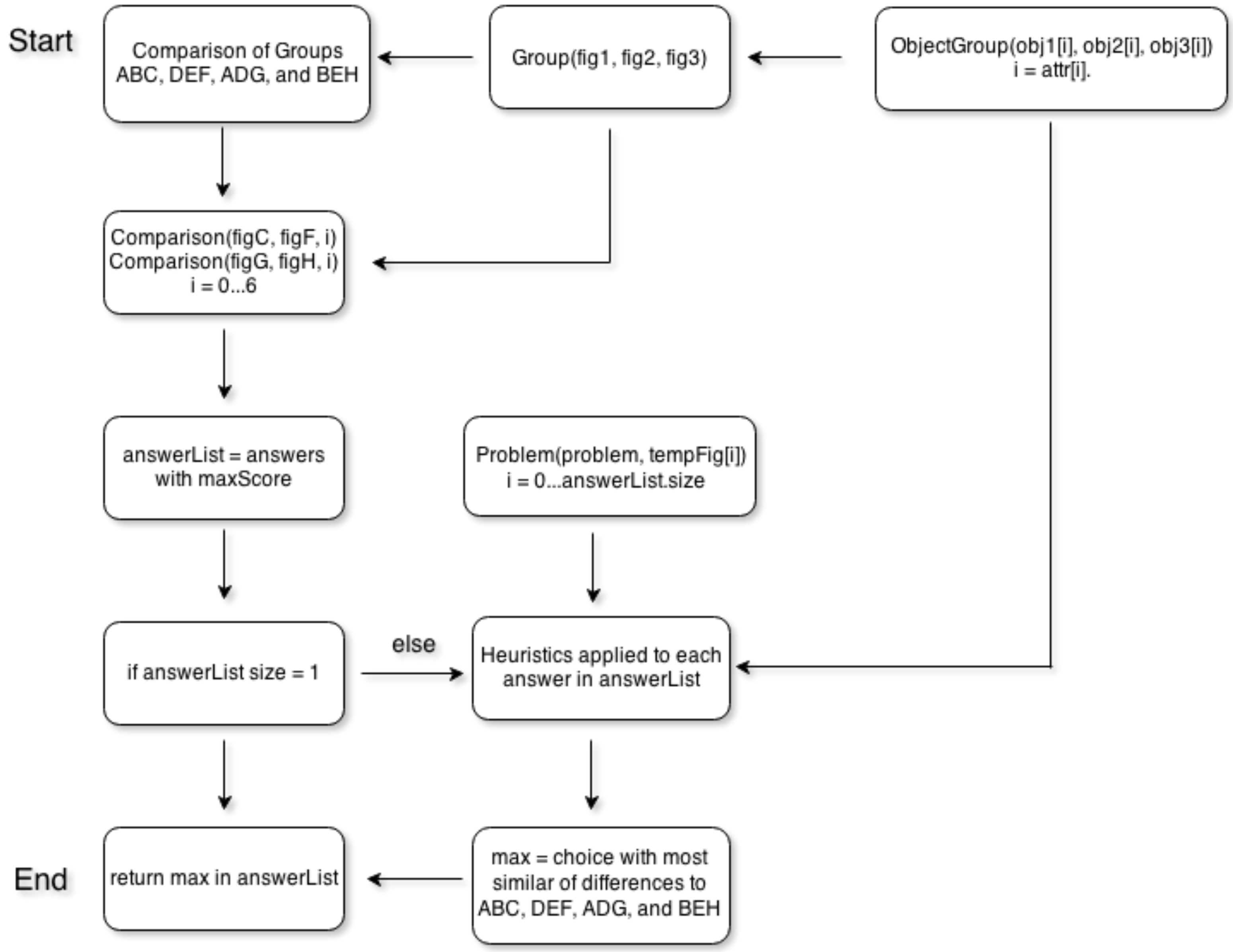
The purpose of this class was to run heuristics on the given problem due to the horizontal and vertical relationships across figures. Then, this was used to compare to the relationship formed from the temporary Raven's Figure passed in as a parameter. This class was a container for library functions that solve leveraged.

Variables

- *problem:* the Ravens Problem passed in as a parameter to the constructor.
- *tempFig*: the Ravens Figure passed in as a parameter to the constructor.
- *figA:* Figure A from the problem passed as the parameter.
- *figB:* Figure B from the problem passed as the parameter.
- *figC:* Figure C from the problem passed as the parameter.
- *figD:* Figure D from the problem passed as the parameter.
- *figE:* Figure E from the problem passed as the parameter.
- *figF:* Figure F from the problem passed as the parameter.
- *figG:* Figure G from the problem passed as the parameter.
- *figH:* Figure H from the problem passed as the parameter.
- *figureList*: array list that contained all of the figures attached to each problem.

Key Functions

- *shapeLeast:* returned true if a shape occurred the least amount of times throughout the figures given and was the same shape as the tempFig.
- *sizeLeast:* returned true if a size occurred the least amount of times throughout the figures given and was the same shape as the tempFig.
- angle*Least:* returned true if an angle occurred the least amount of times throughout the figures given and was the same shape as the tempFig.
- *numObjectsIncrease*: returned true if the number of objects increased horizontally or vertically as you went down the row/column.
- sizeIncrease/sizeDecreased: returned true if the size of each object increased/ decreased horizontally or vertically as you went down the row/column.
- sameShape: returned true if each row and column contained the same shape.
- directionAlt: returned true if the angle/direction of each shape alternated as you went down the row or column.

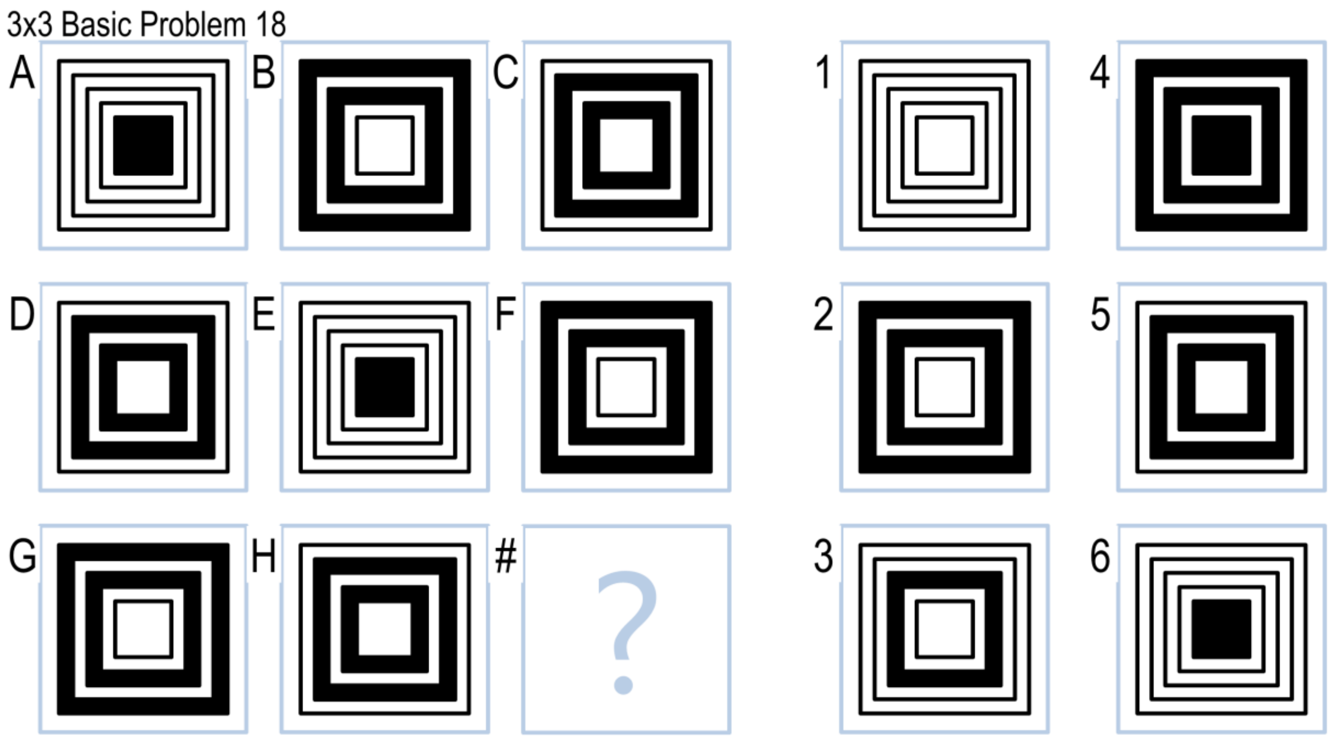


Start
Comparison of Groups ABC, DEF, ADG, and BEH
Group(fig1, fig2, fig3)
ObjectGroup(obj1[i], obj2[i], obj3[i]) i = attr[i].
Comparison(figC, figF, i) Comparison(figG, figH, i) i = 0...6
answerList = answers with maxScore
Problem(problem, tempFig[i]) i = 0...answerList.size
if answerList size = 1
else
Heuristics applied to each answer in answerList
End
return max in answerList
max = choice with most similar of differences to ABC, DEF, ADG, and BEH

Flow Diagram of how Solve finds the Answer

2. Application

We will walk through an example using the strategy I implemented to show how the agent solves a Raven's progressive matrices problem.



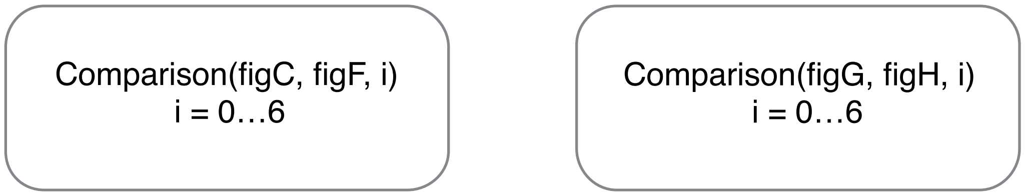
A. Example Problem Walk-Through

Comparison of Groups ABC & DEF

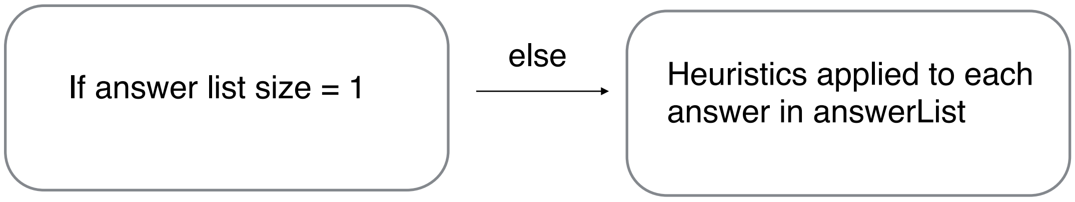
Comparison of Groups ADG & BEH

From these comparisons we will notice that each Group of figures all contains different orders of fill. So when comparing these groups together, we will store something in our hash map like this:

Hash Map of Groups ABC, DEF, ADG, and BEH	
Shape	SAME
Size	SAME
Fill	FALSE - VARIES
Angle	SAME
Inside	SAME
NumShapes	SAME



Therefore, when we run the comparison of all the answer choices, we need to look for fill variations that will alternate and not be repeated. This can be done of our Figure counter, which will make sure that the number of figures is in balance. In this problem, it appears only 3 different RavensFigures are used, and each is used once. Therefore, our heuristics will eliminate figures 2 and 5 from contention, leaving 1, 3, 4 and 6.



Because our answer list will contain 4 values, we will iterate through each choice and apply different heuristic functions on it and the more it passes the higher the score it gets. My heuristics will recognize that there are only 3 different figures used in the problem, where two have appeared three times, and one figure has only appeared twice. It will then look for this figure and find that it is indeed answer choice 6, and will return it as the solution to this problem. The other answer choices were weeded out due to not occurring at all in the given figures for the problem. A table has been listed below for all of the scores of every figure in the tie list.

Scores of Figure 1, 3, 4 and 6	
Figure 1	10
Figure 3	8
Figure 4	8
Figure 6	14

The main giveaway for our agent is the ability to count the occurrence of objects that are given for figures A through H. Because our agent can recognize when 2 figures are the same, it can then keep track in a hash map of the number of times they appear. It immediately saw that figures 2 and 5 occurred three times and that answer choice 6 had only occurred twice. It inferred then that since figure 6 had occurred the first two rows and columns, that it made a diagonal pattern and belonged as the answer choice.

return answer = “6”

3. Shortcomings

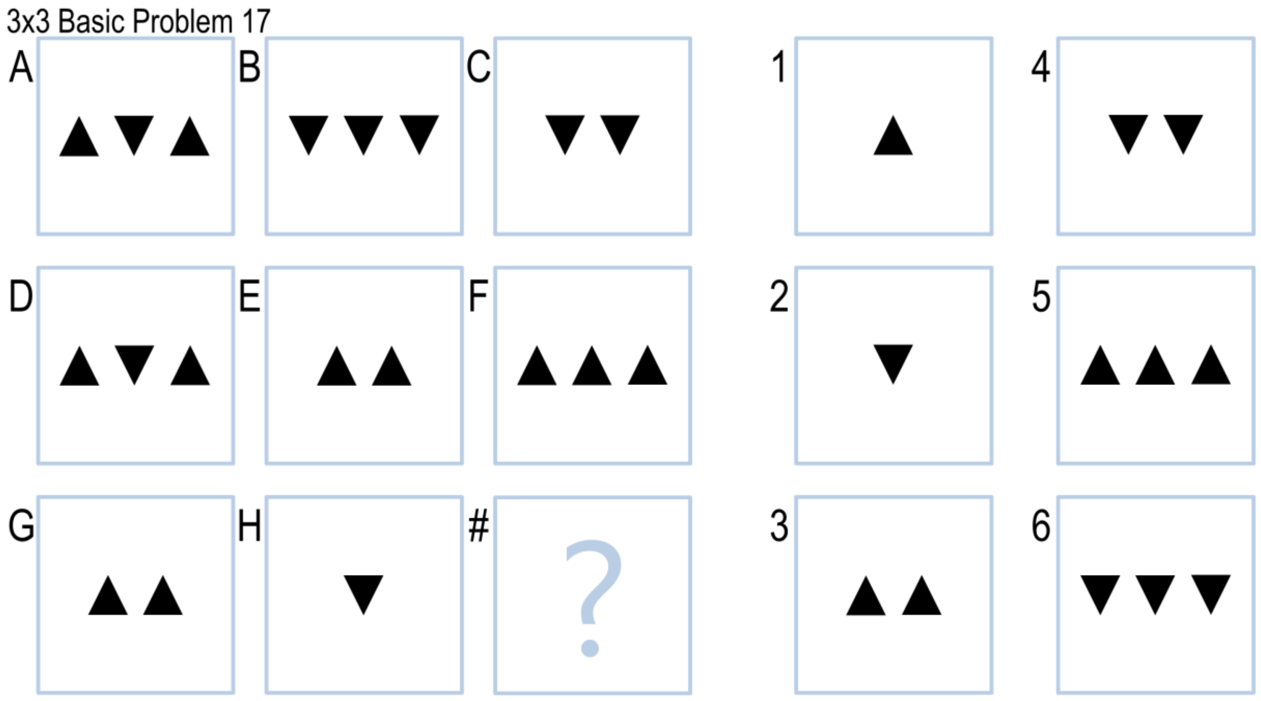
Obviously, this agent does have some downfalls. One of the biggest issues is determining very odd patterns, such as combinations between the number of objects and the variation of each angle in an individual figure. These combinations can be tricky to catch and cause issues for my agent to solve.

A. When does it fall short?

My agent usually fails when there are multiple objects within a figure that are alternating angle changes from each other. These angle changes then carry over to the next horizontal figure, or even the figure above and below (vertical). It’s hard for my agent to recognize this complex of a pattern and be able to categorize an answer choice on that.

B. Why?

I thought it would be easiest to demonstrate this through a short example.



Right now, my agent will heuristically think that figures 3 and 4 are the only possible answers, because statistically the number of objects seems to fluctuate as you travel horizontally and vertically through each row/column. Because figure H only has 1 object and figure F has 3 objects, my agent assumes that the answer probably has 2 objects. It will then eliminate figures 1, 2, 5, and 6 and try to now choose between figures 3 and 4. However, the answer is figure 1, so my agent's heuristics did not take into account the number of objects and angle difference, but just the number of objects. This is a scenario where a complicated relationship between figures is difficult for my agent to recognize.

4. Potential for improvements

A. What could have been done better?

The agent could have been much better organized. Instead of just incrementing the score value based on a few key heuristics, it could have labeled the type of problem based on certain characteristics such what are the main transformations? Not just if the number of object changes, but in what patterns do they change? Is this going to be a factor for determining the answer? Sometimes, simple heuristics will end up eliminating the correct answer because some of the object characteristics just are not a factor when choosing the answer. It is true intelligence when the agent can take in multiple relationships and determine which are relevant and which are white noise. This would have stopped my agent from eliminating the correct answer in the first iteration of scoring.

Another way my agent could have been improved is through learning from earlier problems. I did not incorporate a way for my agent to store problems as frames, so that they could be referenced later as a learning tool. This would have been extremely powerful and useful for the more difficult problems such as number 17.

5. Run-Time Analysis

A. How can it be solved more efficiently?

I have yet to see any real efficiency gaps because of my implementation, but that's mainly because the problem space is so small. For instance when AI agents are trying to play chess, the branching factor can become huge, making efficiency one of the most important things. To analyze how my agent runs the solve function, I am going to break down each of the steps it runs through, and try to quantify the run-time there:

Iterate through each answer choice (figures): *O(n)*

Iterate through each object of each figure: *O(k)*

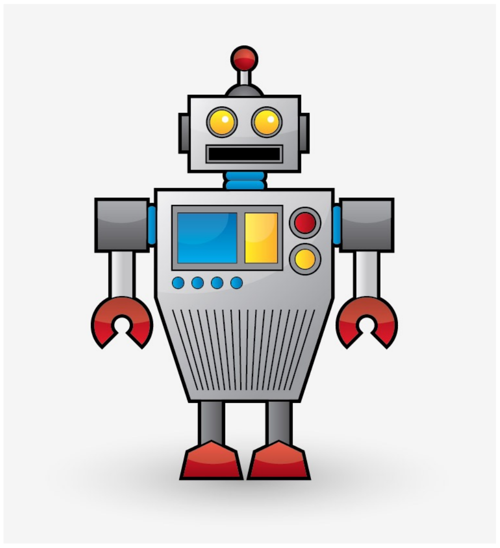
Iterate through each attribute of each object: *O(c)*

Obviously, these steps are run multiple-times throughout solve, but as these numbers approach infinity, addition in big-O goes to 0. So, if I had to estimate the worst-case run-time of my solve algorithm at one point, I would say it's *O(nkc)*, where *n* is the number of answer choices, *k* is the number of objects in each figure, and *c* is the number of attributes for each object. The run-time could become much worse if each figure started to contain many more objects, where each object had many attributes. Right now there are only 6 possible answer choices, but obviously this could be increased as well, hindering the run-time.

This run-time could be improved by storing the data in hash tables and O(1) access variables, eliminating the amount of times that iteration is needed through each level. Some of the data about each figure is calculated more than once as there are multiple rounds of scoring. This duplication is wasted and could be reduced by having a hash table store all of the important information about each figure, once. Then, I could simply index it as it becomes needed throughout solve. However, this extra computation is minor and does not have a great impact on the overall run-time.

6. Relation to Human Cognition

A. How “intelligent” is my agent?



Even though my agent is much more intelligent than my implementation from Project 2 (at least based on the scoring), I would definitely say it has a very long way to go. In some ways it definitely mirrors human-like intelligence. It tries to make quick inferences about the problem and eliminate choices that obviously don’t pass preliminary cases. My agent very much uses problem reduction to analyze the goal situation and quickly eliminate the choices that don’t pass the initial screening. I feel like humans solve problems very much in this same way. For instance, any type of multiple choice exam, I try to eliminate as many answers as possible before even choosing a solution. This is natural as it’s hard to fully concentrate and break down many possible answers, but when working with a much smaller answer space, this becomes much easier. This works the same way for my agent: the more choices it can eliminate the higher the probability it has of choosing the right answer. It also now can recognize patterns for groups of figures, which is a very human-like approach to solving a Raven’s progressive matrices problem.

However, there are some situations where it just cannot compete with human intelligence. Even though my agent has improved on pattern detection, it now needs to determine which information is actually useful for the current problem. Just because it knows that the number of objects differs between figures, that doesn’t mean this problem’s answer choice depends on that data. A human would be able to evaluate and infer from all of the knowledge, seeing that some is more relevant than others, and justly include that when eliminating answers. My agent uniformly scores information the exact same for every single problem, which is not very human-like. This could be improved by allowing my agent to categorize the type of problem it’s facing, and then choose the scoring system that matches the type of problem. This will allow heuristics to be weighted differently depending on how crucial the information is for the problem.