PROJECT REPORT ON :

SOLVING PSYCHOMETRIC TESTS USING RAVEN'S PROGRESSIVE MATRIX AI AGENT

FOR

ARTIFICIAL INTELLIGENCE

CSE - 3013
SLOT - B2+TB2

SUBMITTED BY :

15BCE0673	JAYITHA DEVINENI
15BCE0686	HIMANI JAIN
15BCE0894	GARIMA YADAV
15BCE0906	GEETANJALI TEWARI

SUBMITTED TO :

PROF. ANNAPURNA J.

INDEX

MODULE 1

ACKNOWLEDGEMENT

It gives us immense pleasure to express our deepest sense of gratitude and sincere thanks to our respected and esteemed guide **Prof. Annapurna J.** for her valuable guidance, encouragement and help for completing this work. His useful suggestions for this whole work and co-operative behaviour are sincerely acknowledged. We would also like to thank VIT University , Vellore for encouraging us towards the subject Artificial Intelligence and making a project in it.

MODULE 2

PROBLEM STATEMENT

The Raven’s Progressive Matrices (RPM) test is a commonly used test of general human intelligence. The RPM is somewhat unique as a general intelligence test in that it focuses on visual problem solving, and in particular, on visual similarity and analogy. We are developing a small set of methods for problem solving in the RPM which use propositional, imagistic, and multimodal representations, respectively, to investigate how different representations can contribute to visual problem solving and how the effects of their use might emerge in behaviour.

MODULE 3

INTRODUCTION

[I] MOTIVATION

To check that is when a program is achieving perfect scores on an established and standardized IQ test was created ; would that program truly be intelligent.

[II] SIGNIFICANCE

Giving an answer to the provocative statement above is not trivial: human intelligence and artificial intelligence (AI) have both been given several definitions. Without specifying what is intended by intelligence and whether that is actually measured by IQ tests, it is impossible to answer the question in a proper fashion. Although this project is rooted in computer science and will not discuss the intricacies of cognitive psychology and the human brain, we will start by providing a quick overview of what intelligence and artificial intelligence are . It will examine related previous work, describe the similarities and differences of our approach and discuss some key concepts about the task of solving RPM before going into the details of our implementation.

[III] SCOPE

The scope of this project is :

- Solving as many RPM problems as possible from the three hardest sections of the problems.
- Mapping human intelligence to artificial intelligence.

[IV] APPLICATIONS

Solving RPM is a measure for determining intelligence of machines as well as humans , we can use this on people suffering from Asperger's Disorder and test their special abilities . It can be the official measure to give a score of the IQ of any agent.

MODULE 4
LITERATURE SURVEY

The Raven’s Progressive Matrices (RPM) test [1] is a standardized intelligence test that consists of visually presented, geometric-analogy-like problems in which a matrix of geometric figures is presented with one entry missing, and the correct missing entry must be selected from a set of answer choices. [2]

Although the test is supposed to measure only educative ability, or the ability to extract and understand information from a complex situation (Raven, Raven, & Court),[3]the RPM’s high level of correlation with other multi domain intelligence tests have given it a position of centrality in the space of psychometric measures (Snow , Kyllonen, & Marshalek), and it is therefore often used as a test of general intelligence. [4]

Using the RPM as a measure of general intelligence, though it consists only of problems in a single, nonverbal format, stands in contrast to using broader tests like the Wechsler scales,[5] which are comprised of subtests across several different verbal and nonverbal domains. Despite its widespread use, neither the computational nor the cognitive characteristics of the process of solving the RPM are well understood. [6]

Hunt gives a theoretical account of the information processing demands of certain problems from the Advanced Progressive Matrices (APM),[7] in which he proposes two qualitatively different solution algorithms—“Gestalt,” which uses visual representations and perceptually based operations, and “Analytic,”[8] which uses feature-based representations and logical operations—that could yield identical results on at least portions of the test. Our work expands on Hunt’s idea by asking whether qualitatively different systems of representation can lead to identical performance on the RPM. [9].Our question is theoretically interesting for the study of cognition and AI because the fact that the RPM correlates so well with broader tests of intelligence suggests that the specific information processing capacities tapped by the RPM may be central to domain-general processes of reasoning[10] used across a variety of cognitive tasks. If different schemes of representation can provide equivalent performance, then this raises the issues of

1) Whether these broader reasoning processes can (or must) also be instantiated with different types of representations,[11]

2)To what extent a single individual may (or must) draw on these different representations for reasoning tasks, and [12]

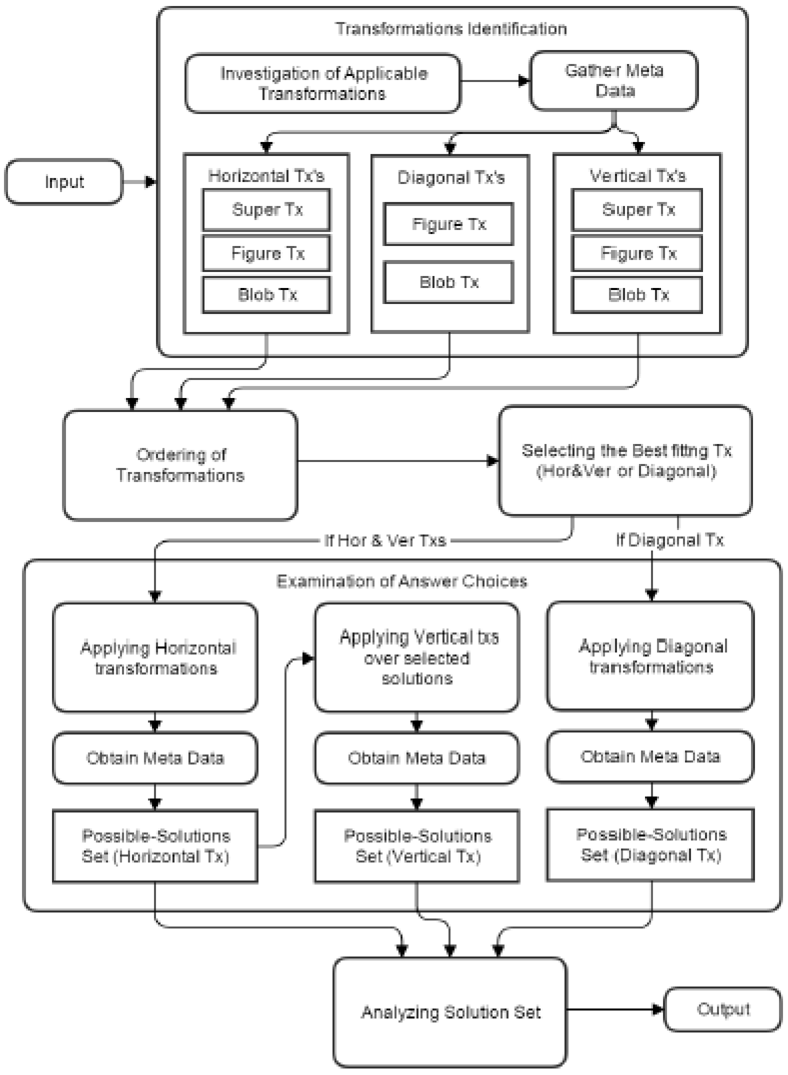
3) To what degree individual differences may (or must) result from variations in the underlying representations being used.[13]

Our central question is also of immense practical import because the RPM family of tests is used extensively in clinical, educational, vocational, and scientific settings as an accurate assessment of intelligence.[14] Therefore, interpretations of RPM scores should be made only with a thorough understanding of what cognitive implications these scores do and do not provide[15]

MODULE 5

Implementation

[I] Architecture



Transformations Identification
Investigation of Applicable Transformations
Gather Meta Data
Input
Horizontal Tx's
Super Tx
Figure Tx
Blob Tx
Diagonal Tx's
Figure Tx
Blob Tx
Vertical Tx's
Super Tx
Figure Tx
Blob Tx
Ordering of Transformations
Selecting the Best fitting Tx (Hor&Ver or Diagonal)
If Hor & Ver Txs
If Diagonal Tx
Examination of Answer Choices
Applying Horizontal transformations
Applying Vertical txs over selected solutions
Applying Diagonal transformations
Obtain Meta Data
Obtain Meta Data
Obtain Meta Data
Possible-Solutions Set (Horizontal Tx)
Possible-Solutions Set (Vertical Tx)
Possible-Solutions Set (Diagonal Tx)
Analyzing Solution Set
Output

Fig.1] Model

[II] Algorithm

STEP 1 : Agent Design

In order to find transformations, the agent looks for super transformations which occur over an entire row across all figures. This is considered as the highest level of abstraction. Then, the agent looks for figure transformations which repeat itself in a row between two figures. If the agent is not able to find any pattern at the figure level, then the agent looks for blob transformations which occur at the object level between two figures.

STEP 2 : Input

Visual images are fed as input to the agent. For a 3x3 RPM problem, 8 problem figures and the 8 answer options are given as input. The AI agent reads the images and converts them from RGB image to bi-level image that consists of a single color band with black and white pixel data.

STEP 3 : Transformation Identification

Three different set of transformations are identified namely, horizontal, vertical and diagonal. The first row of figures is used to find the horizontal transformation. The main diagonal is used to find the diagonal transformation. And the first column of figures is used to find the vertical transformation. In order to find the possible transformations 3 images are fed as input. The images are first checked for super transformations. Super transformations include Constant Addition, Constant Subtraction, Addition (The third image is the addition of first two images), Subtraction (The third image is the difference of the first two images), Common (The third image is the common pixels present in the first two images), Divergence, Convergence and Migration (Objects in first image move in different directions over the next two images) such that they form a moving sequence .

Each of the transformations returns a confidence score out of 100 which is an indicator of the applicability of that transformation. Also, for each transformation, meta data is recorded and they depend on the nature of the transformation. Meta data includes the offset of images, added or subtracted area, point of convergence, similarity measure, migration distance, etc. A Transformation Frame is used to represent all these data in a meaningful form. Based on the score, the transformation frame keeps track of the most probable transformation as when details of new transformations are added.

Next, the figure level transformations are checked. Each figure transformation takes 2 images as input and gives a confidence score and meta data for later analysis. Figure level transformations include, Similarity (The two images are similar), Repetition by Expansion (An object in the first image repeats itself again and expands) and Repetition by Translation (An object in the first image repeats and moves some distance)
Blob level transformations are investigated only if none of the above transformations return a confidence score greater than the threshold (=99). The agent dives deeper and looks for blob level transformations such as morphing, translation of blobs, scaling etc.

STEP 4 : Blob Level Transformation

Blobs are nothing but a group of pixels that represent a shape. Each blob is a fully connected component, meaning, a blob cannot have two disjoint components. Given an image, blobs are extracted by using bread first search technique. The bounding box of the active pixels is obtained. A pixel can be considered as active if it is a part of an object/blob in the image. The pixels array is searched for the first active pixel. Once first active pixel is found the, a breadth first search is performed to convert all the active pixels to inactive. Thus a single breadth first search deletes a single object from the figure. During this process, the coordinates of the object, height width, number of filled pixels, fill ratio, center, etc are calculated. The above process is repeated to find the details of all the objects in the image. Once details of the blobs in both the given images are gathered, correspondence between the blobs in one image to other is found. Corresponding blobs are identified by matching their center, fill ratio, dimensions and their coordinates in the image. Once they are calculated, data of the blobs are analyzed to extract meta data such as one-to-one corresponded of objects in the images, repetition or addition of new objects in the images, deletion of objects etc. Any morphing pattern if found is also made not of. The change in the number of objects between images is also made not of. This meta data is further used to decide the relationship between the given images. Some of the blob transformations include, Morphing (Presence of shape change), Count difference (Pattern based change in the number of blobs), Scaling of One Object, Translation of One Object.
Thus similar to humans, blob level transformations are checked by the agent only if it is not able to find any applicable transformations at higher levels.

STEP 5 : Ordering of Transformations

After the transformations are found, they are ordered based on their confidence scores, in such a way that the transformations with highest probability of occurrence are tested first with the answer choices in the next step.

STEP 6 : Selecting the Transformation Set

Based on the confidence score of the Diagonal and Horizontal Transformation, this module chooses which to consider while examining answer choices. If diagonal transformation set has the highest confidence score, then only the main diagonal is considered to find the solution.

STEP 7 : Examining Answer Choices

In case of diagonal transformation, the chosen set is tested on the answer choices along with the main diagonal images A and E. The ones that satisfy the transformation criteria are added to the solution set and forwarded for further analysis to the next module.
In case of horizontal transformation set, the ordered transformations are tested for images in the last row (G, H, I) where I is replaced with each answer choice. For each transformation scores and meta data are recorded. If the values obtained are within permissible range when compared to original values of the first row transformations, then the answer choice is added to the solutions set.
The answer choices in the computed solutions set is then examined for vertical transformations. The resulting solution set might or might not contain a single answer. So, both the solution sets are sent for further analysis.

STEP 8 : Analyzing Solution Set

If the vertical solution set contains only one answer choice, then it is returned as the solution to the problem. If the vertical solution set contains multiple probably answer choices, then they are sorted based on their deviation from the original scores and the answer choice with the least deviation is returned as the answer. If the vertical solution set is empty then the answer with least deviation in the horizontal solution set is returned as the answer. If both the sets are empty, then this suggests that the agent couldn't decide on an answer choice and so it skips the problem.

STEP 9 : Output

The output returned is an integer indicating the number of the answer choice.

[III] Complexity analysis

The complexity of the algorithm increases with the toughness of the question , i.e. when there is a tougher problem then the probability of the agent to give the correct answer is very less. The agent is highly accurate in predicting the solutions for most of the basic problems. The layered abstraction approach makes the agent very efficient in terms of time spent on solving a problem. The agent spends less time (1 or 2 seconds) for solving easier problems. For complex problems the longest duration taken to solve by the agent was 10 seconds.

[IV] Coding Implementation

```
from PIL import Image
import itertools
import collections
from RavensProblem import RavensProblem
from RavensFigure import RavensFigure
from RavensObject import RavensObject
import pprint
import os
from NumberOfIslands import Islands
import functools
import statistics

class Agent:
    def __init__(self):
        self.problem_name = None
        self.log_file = None
        self.figure_names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', '1', '2', '3', '4', '5', '6', '7', '8']
        self.figure_pixel_matrix = {}
        self.figure_similarity = {}
        self.figure_objects = {}
        self.islands = Islands()

    def initialize(self, problem: RavensProblem):
        self.figure_objects = {}
        print(problem.name)
        self.problem_name = problem.name
        folder_name = 'Problems/{}s {}/{}/'.format(problem.name.rsplit(' ', 1)[0], problem.name.split('-')[0].rsplit(' ', 1)[1], problem.name)
        for figure_name in self.figure_names:
            self.figure_pixel_matrix[figure_name] = self.open_image_return_pixels(Image.open('{}{}.png'.format(folder_name, figure_name)).convert('1'))
        for combination in [['A', 'B'], ['B', 'C'], ['D', 'E'], ['E', 'F'], ['G', 'H'], ['H', '1'], ['H', '2'], ['H', '3'], ['H', '4'], ['H', '5'], ['H', '6'], ['H', '7'], ['H', '8']]:
            self.figure_similarity[combination[0] + combination[1]] = self.calculate_figure_pixel_similarity(self.figure_pixel_matrix[combination[0]], self.figure_pixel_matrix[combination[1]])


    def calculate_figure_pixel_similarity(self, pixel_matrix_a, pixel_matrix_b):
        union = 0
        intersection = 0
        for (pixel_a, pixel_b) in zip(functools.reduce(lambda x, y: x + y, pixel_matrix_a), functools.reduce(lambda x, y: x + y, pixel_matrix_b)):
            union += 1
            if pixel_a == pixel_b:
                intersection += 1

    def figure_pixel_matrix_white_union(self, pixel_matrix_a, pixel_matrix_b):
        return_matrix = [[0 for i in range(len(pixel_matrix_a[0]))] for j in range(len(pixel_matrix_a))]
        for row, (a_row, b_row) in enumerate(zip(pixel_matrix_a, pixel_matrix_b)):
            for column, (a_row_column, b_row_column) in enumerate(zip(a_row, b_row)):
                if (a_row_column | b_row_column) != 0:
                    return_matrix[row][column] = 255
                else:
                    return_matrix[row][column] = 0
        return return_matrix

    def figure_pixel_matrix_black_union(self, pixel_matrix_a, pixel_matrix_b):
        return_matrix = [[0 for i in range(len(pixel_matrix_a[0]))] for j in range(len(pixel_matrix_a))]
        for row, (a_row, b_row) in enumerate(zip(pixel_matrix_a, pixel_matrix_b)):
            for column, (a_row_column, b_row_column) in enumerate(zip(a_row, b_row)):
                if a_row_column == 0 or b_row_column == 0:
                    return_matrix[row][column] = 0
                else:
                    return_matrix[row][column] = 255
        return return_matrix

    def figure_pixel_matrix_black_intersection(self, pixel_matrix_a, pixel_matrix_b):
        return_matrix = [[0 for i in range(len(pixel_matrix_a[0]))] for j in range(len(pixel_matrix_a))]
        for row, (a_row, b_row) in enumerate(zip(pixel_matrix_a, pixel_matrix_b)):
            for column, (a_row_column, b_row_column) in enumerate(zip(a_row, b_row)):
                if a_row_column == 0 and b_row_column == 0:
                    return_matrix[row][column] = 0
                else:
                    return_matrix[row][column] = 255
        return return_matrix

    def figure_pixel_matrix_black_xor(self, pixel_matrix_a, pixel_matrix_b):
        return_matrix = [[0 for i in range(len(pixel_matrix_a[0]))] for j in range(len(pixel_matrix_a))]
        for row, (a_row, b_row) in enumerate(zip(pixel_matrix_a, pixel_matrix_b)):
            for column, (a_row_column, b_row_column) in enumerate(zip(a_row, b_row)):
                if a_row_column != b_row_column:
                    return_matrix[row][column] = 0
                else:
                    return_matrix[row][column] = 255
        return return_matrix

    def figure_pixel_matrix_white_intersection(self, pixel_matrix_a, pixel_matrix_b):
        return_matrix = [[0 for i in range(len(pixel_matrix_a[0]))] for j in range(len(pixel_matrix_a))]
        for row, (a_row, b_row) in enumerate(zip(pixel_matrix_a, pixel_matrix_b)):
            for column, (a_row_column, b_row_column) in enumerate(zip(a_row, b_row)):
                if a_row_column == b_row_column and a_row_column != 0:
                    return_matrix[row][column] = 255
                else:
                    return_matrix[row][column] = 0
        return return_matrix

    def parse_objects_in_figures(self):
        for figure_name in self.figure_names:
            self.figure_objects[figure_name]                                                    =
self.islands.get_number_of_islands(self.figure_pixel_matrix[figure_name])
            for key, this_object in self.figure_objects[figure_name].items():
                center_x = int((this_object['left'] + this_object['right']) / 2)
                center_y = int((this_object['up'] + this_object['down']) / 2)
                if (center_y, center_x) in this_object['pixel_location']:
                    this_object['fill'] = 'yes'
                else:
                    this_object['fill'] = 'no'
                status_of_center = 0 if this_object['fill'] == 'yes' else 255
                current_x = center_x
                current_y = center_y
                while True:
                    current_x += 1
                    if current_x > 184:
                        this_object['shape'] = 'unknown'
                        break
                    if (this_object['fill'] == 'no' and (current_y, current_x) in this_object['pixel_location'] and
self.figure_pixel_matrix[figure_name][current_y][current_x] != status_of_center) \
                            or   (this_object['fill']   ==   'yes'   and   (current_y,   current_x)   not   in
this_object['pixel_location']):
                        break
                    current_y += 1
                    if (this_object['fill'] == 'no' and (current_y, current_x) in this_object['pixel_location'] and
self.figure_pixel_matrix[figure_name][current_y][current_x] != status_of_center) \
                            or   (this_object['fill']   ==   'yes'   and   (current_y,   current_x)   not   in
this_object['pixel_location']):
                        break
                r = (this_object['right'] - this_object['left']) / 2
                if r == 0:
                    this_object['shape'] = 'unknown'
                elif 0.4 < (current_x - center_x) / r < 0.6:
                    this_object['shape'] = 'diamond'
                elif 0.6 <= (current_x - center_x) / r <= 0.8:
                    this_object['shape'] = 'circle'
                elif 0.9 < (current_x - center_x) / r < 1.2:
```

```
                this_object['shape'] = 'square'
            else:
                this_object['shape'] = 'unknown'
        return -1

    def open_image_return_pixels(self, image: Image.Image):
        pixels = list(image.getdata())
        width, height = image.size
        return [pixels[i * width:(i + 1) * width] for i in range(height)]

    def method_unchanged(self):
        flag = 1
        best_answer = -1
        similarity_maximum = 0
        for combination in [['A', 'B'], ['B', 'C'], ['D', 'E'], ['E', 'F'], ['G', 'H']]:
            if self.figure_similarity[combination[0] + combination[1]] < 98:
                flag = 0
        if flag == 1:
            for combination in [['H', '1'], ['H', '2'], ['H', '3'], ['H', '4'], ['H', '5'], ['H', '6'], ['H', '7'], ['H', '8']]:
                this_similarity = self.figure_similarity[combination[0] + combination[1]]
                if this_similarity > similarity_maximum:
                    best_answer = combination[1]
                    similarity_maximum = this_similarity
        return best_answer

    def method_merge_row(self):
        best_answer = -1
        maybe_answers = {}
        if self.calculate_figure_pixel_similarity(

self.figure_pixel_matrix_white_intersection(self.figure_pixel_matrix_white_intersection(self.figure_p
ixel_matrix['A'], self.figure_pixel_matrix['B']), self.figure_pixel_matrix['C']),

self.figure_pixel_matrix_white_intersection(self.figure_pixel_matrix_white_intersection(self.figure_p
ixel_matrix['D'], self.figure_pixel_matrix['E']), self.figure_pixel_matrix['F'])) > 98:
            for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                this_similarity = self.calculate_figure_pixel_similarity(

self.figure_pixel_matrix_white_intersection(self.figure_pixel_matrix_white_intersection(self.figure_p
ixel_matrix['A'], self.figure_pixel_matrix['B']), self.figure_pixel_matrix['C']),

self.figure_pixel_matrix_white_intersection(self.figure_pixel_matrix_white_intersection(self.figure_p
ixel_matrix['G'], self.figure_pixel_matrix['H']), self.figure_pixel_matrix[possible_answer]))
                if this_similarity > 97.5:
                    maybe_answers[possible_answer] = this_similarity
            if len(maybe_answers) != 0 and max(maybe_answers.items(), key=lambda x: x[1])[1] > 99:
                return max(maybe_answers.items(), key=lambda x: x[1])[0]
            if len(maybe_answers) == 1:
                best_answer, dumb = maybe_answers.popitem()
            elif len(maybe_answers) > 1:
                similarity_xor = {}
                for key, value in maybe_answers.items():
                    similarity_xor[key]                                                          =
self.calculate_black_area_in_figure(self.figure_pixel_matrix_black_xor(self.figure_pixel_matrix[key]
, self.figure_pixel_matrix['E']))
```

```
                return min(similarity_xor.items(), key=lambda x: x[1])[0]
        return best_answer

    def method_simple_iterate(self):
        possible_combinations = self.permutations(['A', 'B', 'C'], ['D', 'E', 'F'])
        flag = 0
        best_answer = -1
        similarity_maximum = 0
        for combination in possible_combinations:
            overall_similarity = []
            for key, value in combination.items():

overall_similarity.append(self.calculate_figure_pixel_similarity(self.figure_pixel_matrix[key],
self.figure_pixel_matrix[value]))
            if statistics.mean(overall_similarity) > 98:
                flag = 1
        if flag == 1:
            for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                possible_combinations = self.permutations(['A', 'B', 'C'], ['G', 'H', possible_answer])
                for combination in possible_combinations:
                    overall_similarity = []
                    for key, value in combination.items():

overall_similarity.append(self.calculate_figure_pixel_similarity(self.figure_pixel_matrix[key],
self.figure_pixel_matrix[value]))
                    if statistics.mean(overall_similarity) > similarity_maximum and
statistics.mean(overall_similarity) > 98:
                        best_answer = possible_answer
                        similarity_maximum = statistics.mean(overall_similarity)
        return best_answer

    def method_merge_two_to_get_third(self):
        best_answer = -1
        similarity_maximum = 0
        possible_combinations = list(itertools.permutations('012', 3))
        imitation = ['A', 'B', 'C']
        for combination in possible_combinations:
            if
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_union(self.figure_pixel_matrix[
imitation[int(combination[0])]], self.figure_pixel_matrix[imitation[int(combination[1])]]]),
self.figure_pixel_matrix[imitation[int(combination[2])]]]) > 98:
                for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                    reproduce = ['G', 'H', possible_answer]
                    this_similarity =
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_union(self.figure_pixel_matrix[
reproduce[int(combination[0])]], self.figure_pixel_matrix[reproduce[int(combination[1])]]]),
self.figure_pixel_matrix[reproduce[int(combination[2])]]])
                    if this_similarity > similarity_maximum:
                        best_answer = possible_answer
                        similarity_maximum = this_similarity
        return best_answer

    def method_and_two_to_get_third(self):
        best_answer = -1
        similarity_maximum = 0
        if
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_intersection(self.figure_pixel_m
atrix['A'], self.figure_pixel_matrix['B']), self.figure_pixel_matrix['C']) > 97 \
                and
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_intersection(self.figure_pixel_m
atrix['D'], self.figure_pixel_matrix['E']), self.figure_pixel_matrix['F']) > 97:
            for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                this_similarity                                                                    =
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_intersection(self.figure_pixel_m
atrix['G'], self.figure_pixel_matrix['H']), self.figure_pixel_matrix[possible_answer])
                if this_similarity > similarity_maximum:
                    best_answer = possible_answer
                    similarity_maximum = this_similarity
        return best_answer

    def method_xor_two_to_get_third(self):
        best_answer = -1
        similarity_maximum = 0
        if len(self.figure_objects['A']) + len(self.figure_objects['B']) == len(self.figure_objects['C']) and
len(self.figure_objects['D']) + len(self.figure_objects['E']) == len(self.figure_objects['F']):
            return -1
        if
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_xor(self.figure_pixel_matrix['A'
], self.figure_pixel_matrix['B']), self.figure_pixel_matrix['C']) > 97 \
                and
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_xor(self.figure_pixel_matrix['D'
], self.figure_pixel_matrix['E']), self.figure_pixel_matrix['F']) > 97:
            for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                this_similarity                                                                    =
self.calculate_figure_pixel_similarity(self.figure_pixel_matrix_black_xor(self.figure_pixel_matrix['G'
], self.figure_pixel_matrix['H']), self.figure_pixel_matrix[possible_answer])
                if this_similarity > similarity_maximum:
                    best_answer = possible_answer
                    similarity_maximum = this_similarity
        return best_answer

    def method_change_of_area_to_get_third(self):
        best_answer = -1
        similarity_maximum = 0
        first_row = ['A', 'B', 'C']
        second_row = ['D', 'E', 'F']
        possible_combinations = self.permutations(first_row, second_row)
        similarity_of_combination = {}
        for combination in possible_combinations:
            area_difference = []
            correspondence = list(combination.items())

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure(corresp
ondence[0][0]) - self.calculate_sum_of_area_in_figure(correspondence[1][0]),
                                                             self.calculate_sum_of_area_in_figure(correspondence[0][1])
- self.calculate_sum_of_area_in_figure(correspondence[1][1])))

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure(corresp
ondence[1][0]) - self.calculate_sum_of_area_in_figure(correspondence[2][0]),
                                                  self.calculate_sum_of_area_in_figure(correspondence[1][1])
- self.calculate_sum_of_area_in_figure(correspondence[2][1])))

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure(corresp
ondence[0][0]) - self.calculate_sum_of_area_in_figure(correspondence[2][0]),
                                                  self.calculate_sum_of_area_in_figure(correspondence[0][1])
- self.calculate_sum_of_area_in_figure(correspondence[2][1])))
            similarity_of_combination[frozenset(combination.items())] = statistics.mean(area_difference)
        most_likely = min(similarity_of_combination.items(), key=lambda x: x[1])
        if most_likely[1] < 0.2:
            similarity_of_combination = {}
            for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                if ('A', 'D') in most_likely[0] and ('B', 'E') in most_likely[0] and ('C', 'F') in most_likely[0]:
                    combination = {'A': 'G', 'B': 'H', 'C': possible_answer}
                elif ('A', 'E') in most_likely[0] and ('B', 'F') in most_likely[0] and ('C', 'D') in most_likely[0]:
                    combination = {'A': possible_answer, 'B': 'G', 'C': 'H'}
                else:
                    return -1
                area_difference = []
                correspondence = list(combination.items())

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure(corresp
ondence[0][0]) - self.calculate_sum_of_area_in_figure(correspondence[1][0]),

self.calculate_sum_of_area_in_figure(correspondence[0][1])                                    -
self.calculate_sum_of_area_in_figure(correspondence[1][1])))

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure(corresp
ondence[1][0]) - self.calculate_sum_of_area_in_figure(correspondence[2][0]),

self.calculate_sum_of_area_in_figure(correspondence[1][1])                                    -
self.calculate_sum_of_area_in_figure(correspondence[2][1])))

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure(corresp
ondence[0][0]) - self.calculate_sum_of_area_in_figure(correspondence[2][0]),

self.calculate_sum_of_area_in_figure(correspondence[0][1])                                    -
self.calculate_sum_of_area_in_figure(correspondence[2][1])))
                similarity_of_combination[possible_answer] = statistics.mean(area_difference)
            return min(similarity_of_combination.items(), key=lambda x: x[1])[0]
        return -1

    def method_simpler_change_of_area_to_get_third(self):
        area_difference = []

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('A')    -
self.calculate_sum_of_area_in_figure('B'), self.calculate_sum_of_area_in_figure('C')))

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('D')    -
self.calculate_sum_of_area_in_figure('E'), self.calculate_sum_of_area_in_figure('F')))
        similarity_of_answer = {}
        if statistics.mean(area_difference) < 0.1:
            for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                similarity_of_answer[possible_answer] =
self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('G') -
self.calculate_sum_of_area_in_figure('H'), self.calculate_sum_of_area_in_figure(possible_answer))
            filtered_answer = {k: v for k, v in similarity_of_answer.items() if v < 0.1}
            if len(filtered_answer) > 1:
                change_of_length = {}
                for possible_answer, dumb in filtered_answer.items():
                    left_most = 184
                    right_most = 0
                    for index, this_object in self.figure_objects[possible_answer].items():
                        if this_object['left'] < left_most:
                            left_most = this_object['left']
                        if this_object['right'] > right_most:
                            right_most = this_object['right']
                    change_of_length[possible_answer] = right_most - left_most
                return min(change_of_length.items(), key=lambda x: x[1])[0]
            if len(similarity_of_answer) != 0 and min(similarity_of_answer.items(), key=lambda x:
x[1])[1] < 0.1:
                return min(similarity_of_answer.items(), key=lambda x: x[1])[0]
        return -1

    def similarity_of_two_number(self, number_1, number_2):
        number_1 = abs(number_1)
        number_2 = abs(number_2)
        if number_1 + number_2 == 0:
            return 0
        return abs(number_1 - number_2) / ((number_1 + number_2) / 2)

    def calculate_black_area_in_figure(self, matrix):
        return functools.reduce(lambda x, y: x + y, matrix).count(0)

    def calculate_sum_of_area_in_figure(self, figure_name):
        sum_of_area = 0
        for index, figure_object in self.figure_objects[figure_name].items():
            sum_of_area += figure_object['area']
        return sum_of_area

    def method_special_case_in_change_of_area(self):
        best_answer = -1
        if self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('A') -
self.calculate_sum_of_area_in_figure('E'),
                                     self.calculate_sum_of_area_in_figure('F') -
self.calculate_sum_of_area_in_figure('G')) < 0.15:
            similarity_of_combination = {}
            for possible_answer in ['1', '2', '3', '4', '5', '6', '7', '8']:
                area_difference = []

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('E') -
self.calculate_sum_of_area_in_figure(possible_answer),
                                                         self.calculate_sum_of_area_in_figure('G') -
self.calculate_sum_of_area_in_figure('B')))

area_difference.append(self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('A') -
self.calculate_sum_of_area_in_figure(possible_answer),
                                  self.calculate_sum_of_area_in_figure('F') -
self.calculate_sum_of_area_in_figure('B')))
        similarity_of_combination[possible_answer] = statistics.mean(area_difference)
      filtered_similarity_of_combination = {k: v for (k, v) in similarity_of_combination.items() if v
< 0.1}
      if len(filtered_similarity_of_combination) != 0:
        if len(self.figure_objects['A']) == len(self.figure_objects['E']):
          similar_number_of_objects_filtered_of_combination = {k: v for (k, v) in
filtered_similarity_of_combination.items() if len(self.figure_objects[k]) ==
len(self.figure_objects['E'])}
          if len(similar_number_of_objects_filtered_of_combination) == 1:
            key, value = similar_number_of_objects_filtered_of_combination.popitem()
            best_answer = key
    return best_answer

  def method_ugly_number_of_objects(self):
    number_of_objects_diagonal_1 = 0
    number_of_objects_diagonal_2 = 0
    number_of_objects_diagonal_3 = 0
    if len(self.figure_objects['A']) == len(self.figure_objects['E']):
      number_of_objects_diagonal_1 = len(self.figure_objects['A'])
      if len(self.figure_objects['C']) == len(self.figure_objects['D']) == len(self.figure_objects['H']):
        number_of_objects_diagonal_2 = len(self.figure_objects['C'])
        if len(self.figure_objects['B']) == len(self.figure_objects['F']) ==
len(self.figure_objects['G']):
          number_of_objects_diagonal_3 = len(self.figure_objects['B'])
    if not (number_of_objects_diagonal_1 == number_of_objects_diagonal_2 ==
number_of_objects_diagonal_3) and number_of_objects_diagonal_1 != 0 and
number_of_objects_diagonal_2 != 0 and number_of_objects_diagonal_3 != 0:
      # by area
      if self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('A') /
len(self.figure_objects['A']), self.calculate_sum_of_area_in_figure('F') / len(self.figure_objects['F'])) <
0.15 and \
              self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('A') /
len(self.figure_objects['A']), self.calculate_sum_of_area_in_figure('H') / len(self.figure_objects['H']))
< 0.15:
        area_difference = {}
        for possible_answer in [x for x in ['1', '2', '3', '4', '5', '6', '7', '8'] if len(self.figure_objects[x])
== number_of_objects_diagonal_1]:
          area_difference[possible_answer] =
self.similarity_of_two_number(self.calculate_sum_of_area_in_figure('D') /
len(self.figure_objects['D']), self.calculate_sum_of_area_in_figure(possible_answer) /
len(self.figure_objects[possible_answer]))
        if min(area_difference.items(), key=lambda x: x[1])[1] < 0.1:
          return min(area_difference.items(), key=lambda x: x[1])[0]
      # by shape
      same_shape = []
      for possible_answer in [x for x in ['1', '2', '3', '4', '5', '6', '7', '8'] if len(self.figure_objects[x]) ==
number_of_objects_diagonal_1]:
        if self.figure_objects['A']['0']['shape'] != self.figure_objects['E']['0']['shape']:
          if self.figure_objects[possible_answer]['0']['shape'] != self.figure_objects['A']['0']['shape']
and self.figure_objects[possible_answer]['0']['shape'] != self.figure_objects['E']['0']['shape'] and
self.figure_objects[possible_answer]['0']['fill'] == self.figure_objects['A']['0']['fill']:
            same_shape.append(possible_answer)
      if len(same_shape) == 1:
```

```
                return same_shape[0]
            else:
                r_difference = {}
                for possible_answer in same_shape:
                    r_difference[possible_answer] = abs((self.figure_objects['B']['0']['right'] - self.figure_objects['B']['0']['left']) - (self.figure_objects[possible_answer]['0']['right'] - self.figure_objects[possible_answer]['0']['left']))
                if len(r_difference) != 0:
                    return min(r_difference.items(), key=lambda x: x[1])[0]
                else:
                    return -1
        else:
            return -1

    def Solve(self, problem: RavensProblem):
        if 'B-' in problem.name or 'C-' in problem.name or 'Challenge' in problem.name:
            return -1
        self.initialize(problem)
        for possible_method in [self.method_unchanged, self.parse_objects_in_figures, self.method_xor_two_to_get_third, self.method_and_two_to_get_third, self.method_merge_two_to_get_third, self.method_simple_iterate, self.method_merge_row, self.method_change_of_area_to_get_third, self.method_simpler_change_of_area_to_get_third, self.method_special_case_in_change_of_area, self.method_ugly_number_of_objects]:
            possible_answer = possible_method()
            if possible_answer != -1:
                print(possible_method.__name__)
                return possible_answer
        return -1

    def permutations(self, former_objects, later_objects):
        return [dict(zip(a, later_objects)) for a in itertools.permutations(former_objects, len(later_objects))] if \
            len(former_objects) > len(later_objects) else \
            [dict(zip(former_objects, b)) for b in itertools.permutations(later_objects, len(former_objects))]
```

MODULE 6

FUTURE WORK

We have shown that our approach is sufficient for modelling human performance on Raven's Progressive Matrices. An important further step is to use the model to make new discoveries about how people perform spatial problem solving . RPM provides a number of unique opportunities to look at both spatial representation and analogical comparison, due to the complexity and diversity of the problems. By classifying problems based on the model strategies and model components required to solve them, we hope to gain a better understanding of both the factors that make one problem harder than another, and the cognitive abilities that make one person better than another at spatial problem-solving.

MODULE 7

REFERENCES

[1] Barnsley, M. F., & Hurd, L. P. 1992. Fractal Image Compression. Boston: A. K. Peters.

[2] Carpenter, P. A., Just, M. A., & Shell, P. 1990. What one intelligence test measures: A theoretical account of the processing in the Raven Progressive Matrices Test. Psychological Review 97 (3): 404-431.

[3] Davies, J., & Goel, A.K. 2001. Visual analogy in problem solving. In Proceedings of the 17th International Joint Conference on Artificial Intelligence, 377-382. Seattle, WA: Morgan Kaufmann.

[4] Davies, J., Goel, A.K, & Yaner, P.W. 2008. Proteus: A theory of visual analogies in problem solving. KnowledgeBased Systems 21 (7): 636-654.

[5] Dawson, M., Soulières, I., Gernsbacher, M. A., & Mottron, L. 2007. The level and nature of autistic intelligence. Psychological Science 18 (8): 657-662.

[6] DeShon, R. P., Chan, D., & Weissbein, D. A. 1995. Verbal overshadowing effects on Raven's Advanced Progressive Matrices: Evidence for multidimensional performance determinants. Intelligence 21 (2): 135-155.

[7] Dillon, R. F., Pohlmann, J. T., & Lohman, D. F. 1981. A factor analysis of Raven's Advanced Progressive Matrices freed of difficulty factors. Educational and Psychological Measurement 41: 1295–1302.

[8] Hayashi, M., Kato, M., Igarashi, K., & Kashima, H. 2008. Superior fluid intelligence in children with Asperger's disorder. Brain and Cognition 66 (3): 306-310.

[9] Kosslyn, S. M., Thompson, W. L., & Ganis, G. 2006. The Case for Mental Imagery. New York, NY: Oxford University Press.

[10] Kunda, M., & Goel, A.K. 2008a. How Thinking in Pictures can explain many characteristic behaviors of autism. In Proceedings of the 7th IEEE

International Conference on Development and Learning, 304-309. Monterrey, CA.

[11] Law, E., & von Ahn, L. (2011). *Human Computation.* Morgan & Claypool.

[12] Prade, H. & Richard, G. (2011). Analogy-Making for Solv-ing IQ Tests: A Logical View. In *Procs. 19th International Conference on Case-Based Reasoning*, 561-566. London, UK: Springer.

[13] Goel, A., Kunda, M., Joyner, D., & Vattam, S. (2013). Learning about Representational Modality: Design and Programming Projects for Knowledge-Based AI. In *Fourth AAAI Symposium on Educational Advances in Artificial Intelligence*.

[14] Ragni, M. & Neubert, S. (2014). Analyzing Raven's Intel-ligence Test: Cognitive Model, Demand, and Complexity. In H. Prade & G. Richard (Eds.) *Computational Approach-es to Analogical Reasoning: Current Trends*, 351-370. Springer.

[15] McGreggor, K., Kunda, M., & Goel, A. (2014). Fractal and Ravens. *Artificial Intelligence 215*, 1-23.

[16] Goel, A. & Joyner, D. (2015). An Experiment in Teaching Cognitive Systems Online. Technical Report, Georgia In-stitute of Technology.