TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

INTRODUCTION

Manual question generation is one of the most challenging tasks when it comes to facilitating the Massively Online Open Courses (MOOCs) and the exams conducted frequently on a large level. There is a constant need of new questions and generating them manually could be very tough. Along with the considerable time requirement for the generation of questions for an assessment, it is also reported that the development of one quality question for a high-stake test could cost an estimated amount that can range from $1,500 to $2,000[1]. In terms of quality, around 40% of manually constructed questions can fail the performance parameters that were required for their use in assessments [2]. This has highlighted the need of automated question generation. Our project focuses on automated question generation for non-verbal aptitude questions and Multiple Choice Question for a given text. We present a framework which aims on generating various types of questions along with generating meaningful distractors for them.

1.1 Problem Definition

Our project aims to develop a framework that can automate the task of generating nonverbal aptitude questions. We focus on generating questions with an optimal clarity as well as difficulty for the users. We also focus on generating meaningful distractors as they are an important aspect of generating better quality multiple choice questions.

1.2 General Terms

1.2.1 Primitive Shape

This is the unit of a composite figure. Any basic shape like a circle, polygon, straight line, arrow, or a Unicode character could be a primitive shape.

1.2.2 Primitive Rule

These are the basic rules that can be applied to any primitive shape in order to transform it and generate a new primitive shape. Some sample primitive rules are rotate, flip, add vertex, cut etc.

1.2.3 Composite Figure

Any figure made up of one or more primitive shapes is a composite figure. Composite figures are the ones that are used as the question figure.

1.2.4 Composite Rule

A composite rule is a set of one or more primitive rules.

1.2.5 Keyword

This is the correct option to any question.

1.2.6 Distractor

A distractor is the wrong choice for any given question. Ideally a distractor should be similar enough to the key so that the answer is not obvious to the user.

1.2.7 Semantic Role Labelling

It is a task to detect semantic arguments which are associated with the predicate or verbs in the given sentence.

1.2.8 Named Entity Recognition

It is a task to retrieve information which identifies the named entities in a given text into predefined categories like name of the person or location, expressions etc**.**

CHAPTER-2

LITERATURE SURVEY

Automating question generation has been tried for different domains. Since each domain has its own set of requirements for generating a question, the methods involved in question generation of each are also different. In this section, we discuss how similar work has been done in different fields and our learnings from each of them which we could apply in solving our problem statement.

2.1 A Framework for automated generation of questions based on first-order logic

[1] talks about generating questions for specific domains by defining first order logic rules to define question scenarios (arrangement of domain objects) , and answers. The framework can be used to describe any domain using axioms, facts and rules. The framework however does not provide any methodology to validate generated questions – that should be encoded with the domain rules.

For physics, a scenario can be represented by the following figure.

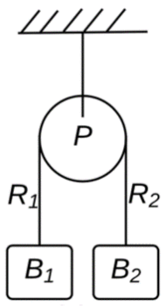
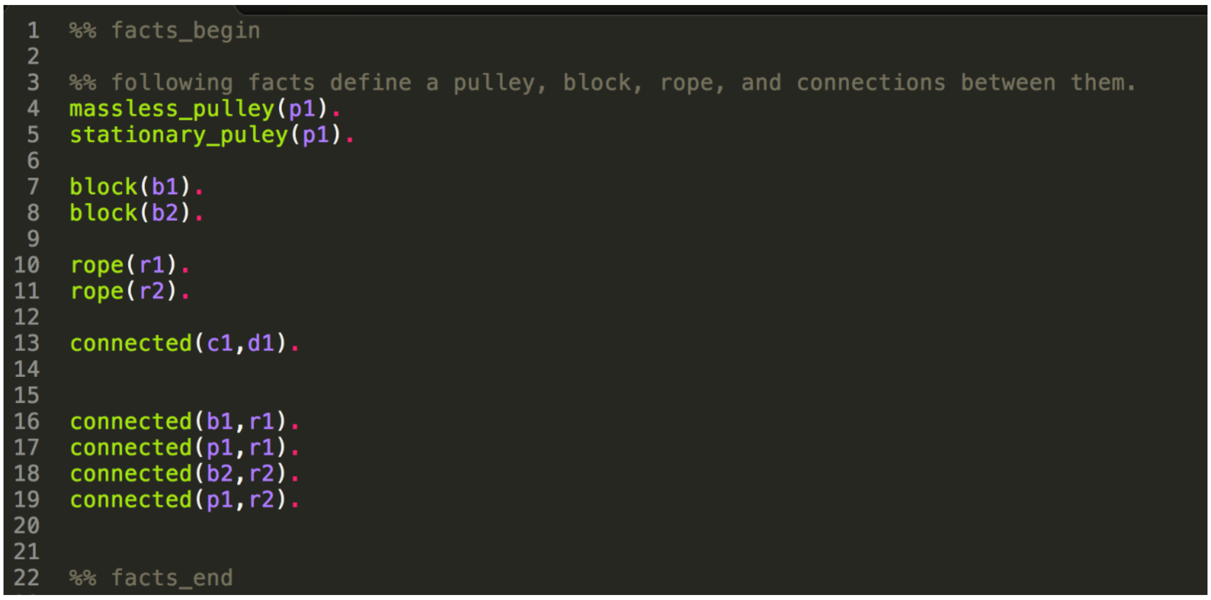


FIG 2.1 A Pulley and a two mass system

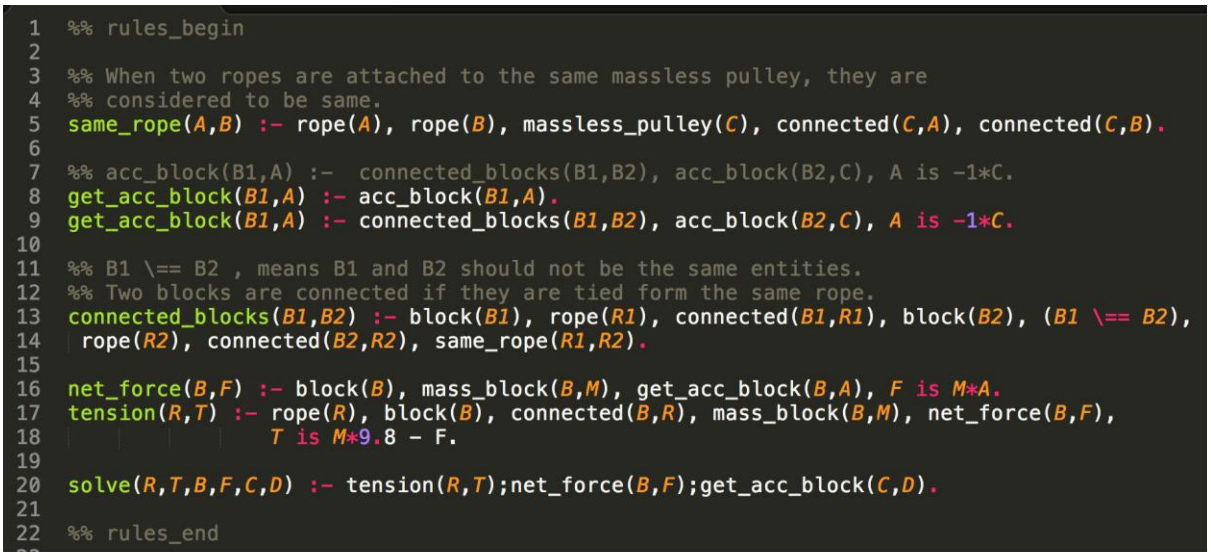
FIG 2.1 consists of two blocks B1 and B2 attached to a massless pulley P. This world can be represented as a collection of logical facts which describe the entities and the relationship between the entities (FIG 2.2).



```
%% facts_begin

%% following facts define a pulley, block, rope, and connections between them.
massless_pulley(p1).
stationary_puley(p1).

block(b1).
block(b2).

rope(r1).
rope(r2).

connected(c1,d1).


connected(b1,r1).
connected(p1,r1).
connected(b2,r2).
connected(p1,r2).


%% facts_end
```

FIG 2.2 Facts defining the entities

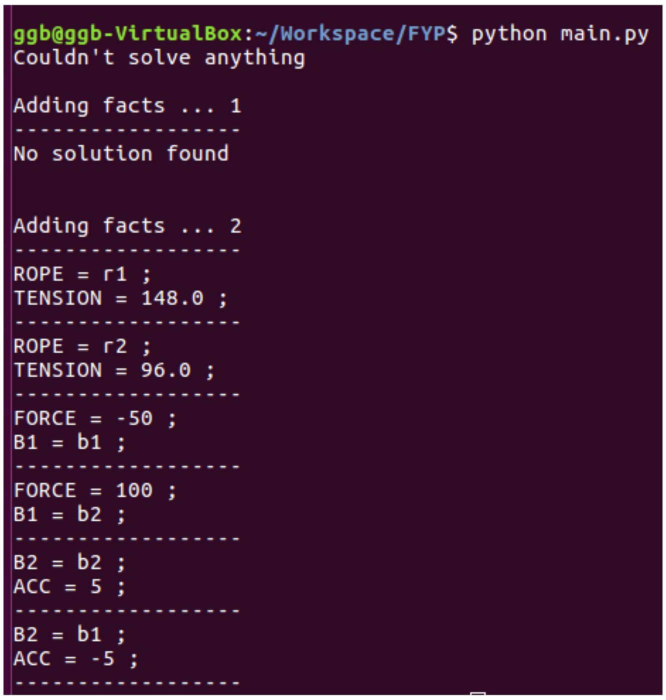
The domain knowledge is coded up as a set of rules that govern world entities. These logical rules use existing rules and facts, to come up with newer facts, which in turn can be fed back into the system. FIG 2.3 shows rules like *same_ropes* (when two ropes will be considered to be the same), *get_acc_block* (how is acceleration of a block calculated) etc.



```
%% rules_begin

%% When two ropes are attached to the same massless pulley, they are
%% considered to be same.
same_rope(A,B) :- rope(A), rope(B), massless_pulley(C), connected(C,A), connected(C,B).

%% acc_block(B1,A) :-  connected_blocks(B1,B2), acc_block(B2,C), A is -1*C.
get_acc_block(B1,A) :- acc_block(B1,A).
get_acc_block(B1,A) :- connected_blocks(B1,B2), acc_block(B2,C), A is -1*C.

%% B1 \== B2 , means B1 and B2 should not be the same entities.
%% Two blocks are connected if they are tied form the same rope.
connected_blocks(B1,B2) :- block(B1), rope(R1), connected(B1,R1), block(B2), (B1 \== B2),
  rope(R2), connected(B2,R2), same_rope(R1,R2).

net_force(B,F) :- block(B), mass_block(B,M), get_acc_block(B,A), F is M*A.
tension(R,T) :- rope(R), block(B), connected(B,R), mass_block(B,M), net_force(B,F),
                T is M*9.8 - F.

solve(R,T,B,F,C,D) :- tension(R,T);net_force(B,F);get_acc_block(C,D).

%% rules_end
```

FIG 2.3 Rules governing the entities

Once the facts and rules are defined, the system can be used to randomly add more entities (with missing values) and use existing facts and rules to find the missing values. The process of finding new values, from existing facts and a set of rules governing those facts mimic the way a student would approach the question. This is the key idea that we pick up from this study.



```
ggb@ggb-VirtualBox:~/Workspace/FYP$ python main.py
Couldn't solve anything

Adding facts ... 1
------------------
No solution found


Adding facts ... 2
------------------
ROPE = r1 ;
TENSION = 148.0 ;
------------------
ROPE = r2 ;
TENSION = 96.0 ;
------------------
FORCE = -50 ;
B1 = b1 ;
------------------
FORCE = 100 ;
B1 = b2 ;
------------------
B2 = b2 ;
ACC = 5 ;
------------------
B2 = b1 ;
ACC = -5 ;
------------------
```

FIG 2.4 Adding facts iteratively until a solution is found

2.2 Automated Generation of High School Geometric Questions Involving Implicit Construction

[2] discusses a combinatorial approach for generating a large set of geometry questions (which require implicit construction for their solution) along with their solutions for a given set of geometry objects. They use a database of theorems and geometric figures, and define an iterative approach for deducing facts from existing facts and rules, to generate valid questions.

As per [2] a geometry question can be represented by a sextuple.

- Objects (lines, triangles, squares, circles etc.)
- Concepts (perpendicular, parallel, midpoint, angle bisector, circumcircle etc.)
- Theorem (Pythagorean Theorem, similarity theorem etc.)
- Construction Objects (perpendicular, parallel, midpoint etc.)
- Relationship (Syntactic, quantitative)
- Query Type (Syntactic, quantitative)

Inputs - Object(s), concept(s), construction object(s), theorem(s) and number of questions to be generated.

Outputs - Question with single or multiple solutions.

The three major components that are described in [2] -

1. GF (Generating Figures) - This component generates geometric figures from the input. The figure constitutes a diagrammatic schema and few unknown variables. Construction Objects (like parallel, perpendicular, midpoint etc.) specified by the user is also added to the figure. Construction objects are the implicit constructions that won't be directly used in question generation.
2. GFS (Generating Facts and Solutions) - This component uses predefined knowledge database of axioms to find values for the unknown variables. This results in formation of a Configuration containing known values for some relationship between its objects. A suitable question is generated from the 'new information' derived from the Configuration. A question is said to be suitable if it covers a new fact and a proper reasoning for the generate fact.
3. GD (Generating data for the figure) - If the question generated does not meet the suitability conditions, then this component assigns values to unknown variables in the Configuration. The more number of times a configuration passes through this component, more and more facts appear as part of the question, resulting in easier

questions. KF and KT are knowledge databases for Figures and Theorems respectively. Both are graph-based knowledge representations.

[2] worked under some constraints. They restricted geometric objects to only triangles and line segments. The components generated from the GF component were restricted only to intersection of two triangles. Finally, the implicated construction should only involve a line segment (parallel, perpendicular or a median) between two existing points and there should be only one such construction.

Algorithm:

1. Generate all possible figures consisting of geometry objects from the given input using the GF function.
2. Add construction object using the GFS component.
3. Keep adding generated figures to KF knowledge database. Proceed after every step only if new figure is formed or else do backtracking.
4. Use GFS to assign values to the variables of the figure using predefined axioms. This results in formation of a configuration.
5. If the configuration meets desired suitability conditions the configuration is the required generated question. If not, the configuration is passed through the GD component to add more data.
6. GD component uses KT (knowledge database of theorems) and assigns values to more unknown variables. If the new configuration generated were not picked before go to step 4, else go to step 3.

We used the concept of starting with a base figure, adding more figures and then applying a set of rules to generate new figures, to generate questions for our problem statement.

2.3 Automatic Multiple Choice Question Generation System for semantic attributes using string similarity measures

Firshoff G and Eskenazi in [3] proposed a system for automatic question generation for vocabulary assessment. They generated 6 types of questions like definition, synonym, antonym, hypernym, hyponym and close questions. They pick the data from WordNet and select a correct sense for it. To generate the distractors, the system selects distractors for the same part of speech with similar frequency. They had generated 60% of theses six types question for the wordlist of 156.

Sung and Lin [5] proposed a system which generates questions from English text Comprehensions. The key features used was to represent the relation between vocabulary and context, for which they used semantic network. The approach was to select adjectives from SemNet of a given text as questionnaire vocabularies and form multiple choice close questions. Correct answer is substituted by the synonyms or antonyms from the WordNet.

Agarwal and Mannem [6] propose a prototype which can generate the gap-filled multiple choice questions. They used lexical and syntactic features for the information extraction from the given sentence, to determine the key and find appropriate distractors. The key word is selected by doing POS (part of speech) tagging. The distractor selection depends on similarity score between the gap filled sentence and the sentence picked up for generating the distractors.

By this literature review we noted that to generate MCQ following steps are needed - the first is to select the sentence which is informative, the second is to choose the appropriate keyword as the correct answer and the last is to find appropriate distractors. Most papers focused on semantic and lexical labels from the given sentence. In our project we chose to extract semantic labels and named entities present in the given sentence. Named entities and semantic labels were very helpful in selecting the keyword. The distractors are picked by applying string matching algorithms.

CHAPTER 3

SYSTEM REQUIREMENT SPECIFICATION

3.1. High Level Block Diagram

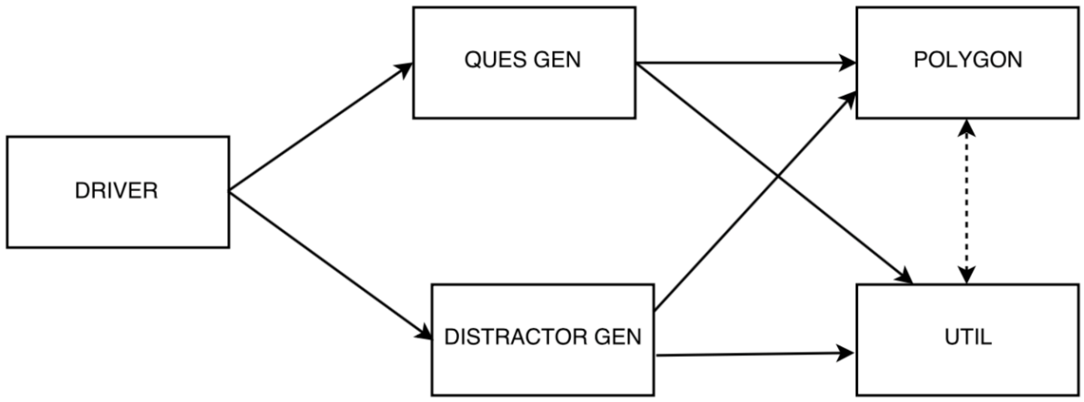


FIG 3.2 High Level Blok Diagram for Non-Verbal Question Generation

FIG 3.1 represents a high level diagram for generating non-verbal aptitude questions. The major components of the framework are a POLYGON class component and a UTIL component. A DRIVER script uses the QUES GEN module and DISTRACTOR GEN module to generate questions and distractors respectively. These are discussed in detail in Chapter-6.

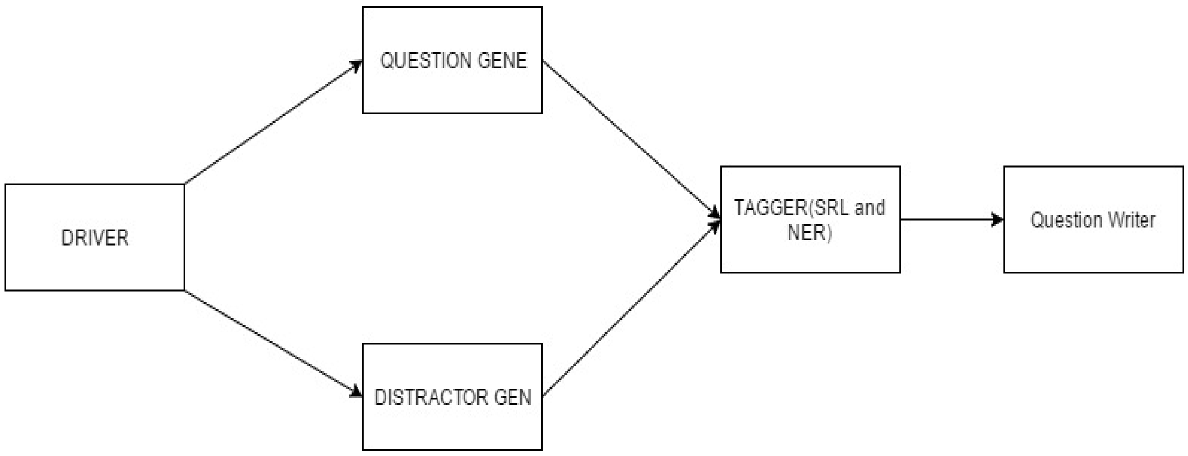


FIG 3.2 High Level Block Diagram for MCQ generation

The basic functionalities are to get SRL and NER labels from the given sentence. The keyword is picked on the basis of the tags obtained from the tagger. The similar sentences are used to generate distractors. The similar sentences are obtained by using string similarity algorithm. If there is no candidate to be the keyword an exception is raised saying that the question is not informative.

3.2. Requirements

3.2.1 Hardware Required

There are no specific hardware requirements.

3.2.2 Software Required

- Python
- Matplotlib
- OpenCV
- Gensim
- PracNLPTools

3.3. Planned Project Solution

3.3.1 Functional Requirements

1. Generate Primitive Figure
    - Should be able to generate a randomized composite figure from a combination of primitive figures
2. Generate composite figure
    - Should be able to generate a randomized composite figure from a combination of primitive figures.
    - Should take parameters like the number of primitive figures used to create the composite figure.
3. Basic operation on figure
    - The framework should allow parameterized operations on the primitive shapes like rotation, resizing, flipping about vertical or horizontal axis, adding or removing a vertex, drawing the figure inside or outside a another existing figure.

- Apart from primitive operations, the framework should provide utility methods to crop an image, concatenate two or more images, draw lines and shapes over the final composite figures. These utility methods are used in generation of questions, key and distractors.

4. Extracting semantic role labels and named entity tags.
   - Should be able to get Semantic role labels tags and Named entities tags in the given sentences
5. Extract keywords from the extracted tags
   - Should be able to figure out the keyword on the basis of named entities and semantic tags obtained
6. Find similar sentences
   - Should be able to find the similar sentences from the database.
7. Utility methods for generating questions
   - Should take all the actual question and replace the keyword with a blank.
   - Should take all the actual question and replace the keyword with a blank.
   - Should take all the keywords and randomly put the options for a given question.

3.3.2 Non Functional Requirements

1. The questions generated should not be trivial, there should be a difficulty level associated with it.
2. The distractors generated should be similar to the Key.
3. The questions generated should be similar to the Key.
4. The question involving multiple components should be valid.
5. The question generated from texts should be meaningful, it should make sense.
6. The question generated from text should resemble the question generated by a human problem setter.

3.4. Constraints and Dependencies

Since matplotlib was used as the tool to generate the figures, the figures which were out of the scope of generation of matplotlib, couldn’t be generated. There are many questions available that involve free-flow shapes and figures. These shapes do not follow any specific rule to be generated and hence were out of our scope of generation. We also had to define different workflows for different types of questions both for generation of questions and for the generation of distractors.

For generating distractors for multiple choice questions we had to choose some of the string similarity algorithms and out of which *gensim's* doc2vec was chosen. Generating distractors from given sentence depends upon the other sentences present in the database. So the database must have most of the sentences belonging to the same domain to generate distractors which are equally probable

3.5. Assumptions

- We assume that if a distractor is composed of the same shapes as that of the key, then it'll look similar to the key.
- We assume that in most cases, the distractors generated will be visibly distinguishable from the key.
- Another assumption that we made was that the questions will not repeat easily as the degree of randomness is multi-level.
- While generating text based MCQ's we have considered only few tags like name of a person, name of an organization, name of the location, dates of any events etc.

3.6. Use Case Diagram

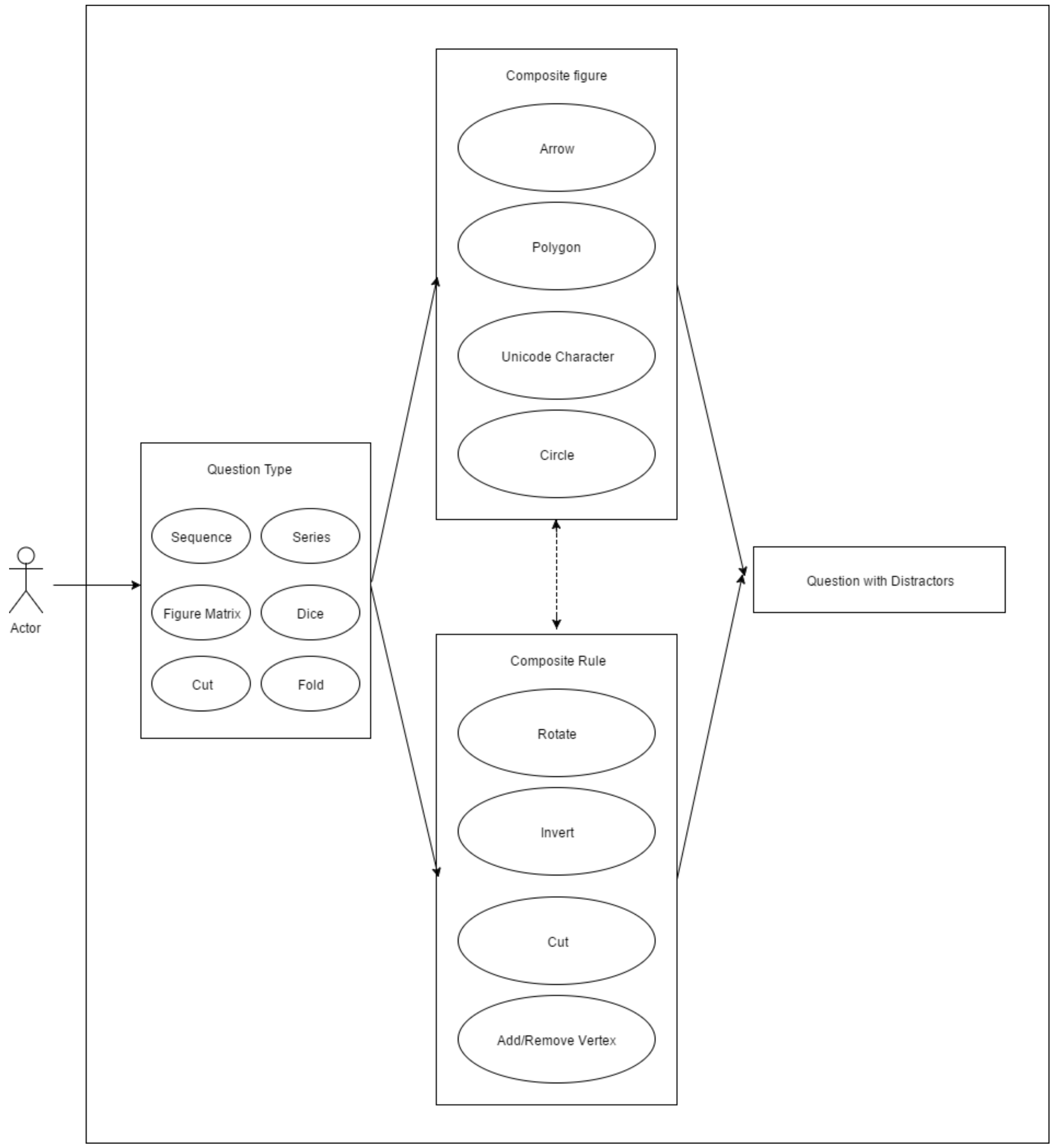


FIG 3.3 Use case diagram

CHAPTER 4

SCHEDULE

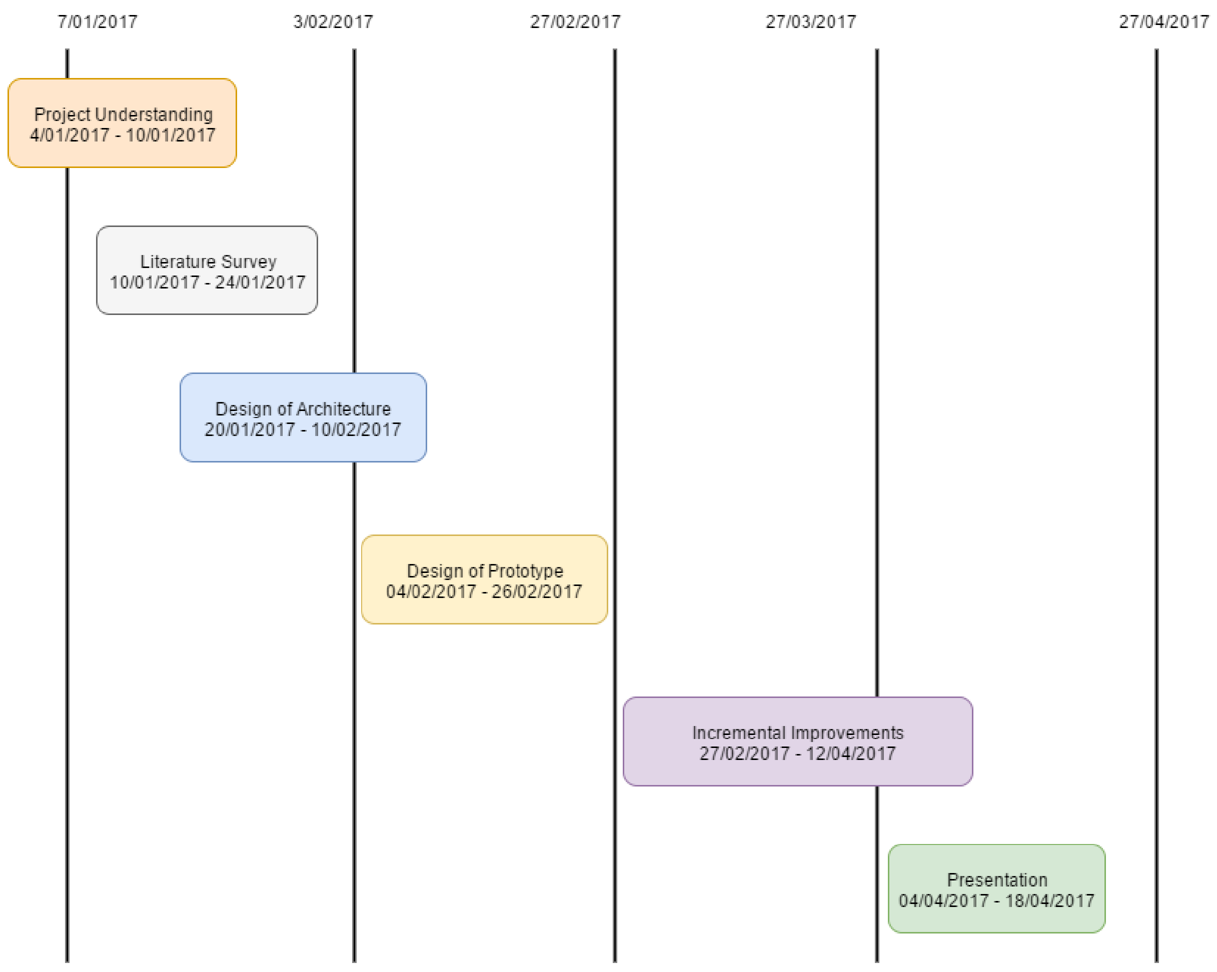

7/01/2017
3/02/2017
27/02/2017
27/03/2017
27/04/2017
Project Understanding 4/01/2017 - 10/01/2017
Literature Survey 10/01/2017 - 24/01/2017
Design of Architecture 20/01/2017 - 10/02/2017
Design of Prototype 04/02/2017 - 26/02/2017
Incremental Improvements 27/02/2017 - 12/04/2017
Presentation 04/04/2017 - 18/04/2017

FIG 4.1 Schedule

During the first month, we spent most of our time on exploring different domains in which automated question generation has been implemented. We also tried to implement [1] in order to understand how primitive rules can be defined to generate questions that can derive solutions using the primitive rules. Once we implemented that, we also explored the importance of distractors in multiple choice questions. After that we took a week to come up with the idea of the framework that could be generic enough to develop a variety of questions. Past that, we first came up with the basic components that'll be required to generate any question. It took us a month to develop the basic structure of the framework and define the primitive rules for polygons to be drawn. We explored turtles and matplotlib as the method to draw the figures and eventually decided to go with matplotlib. Once that was done, it was mostly trying to refine the randomness and adding more modules to try to generate different varieties of questions.

For generating text based MCQ's we implemented many string matching algorithms like Cosine similarity, Dice’s coefficient, Jaccard similarity, and Block distance. We worked on these algorithms for a month and found out these algorithms were not considering the context of the sentence. To consider context we switched to gensim's Word2Vec and explored it. We finally decided to with go gensim's Doc2Vec which gave us better results.

CHAPTER 5

SYSTEM DESIGN

5.1 Architecture Diagram

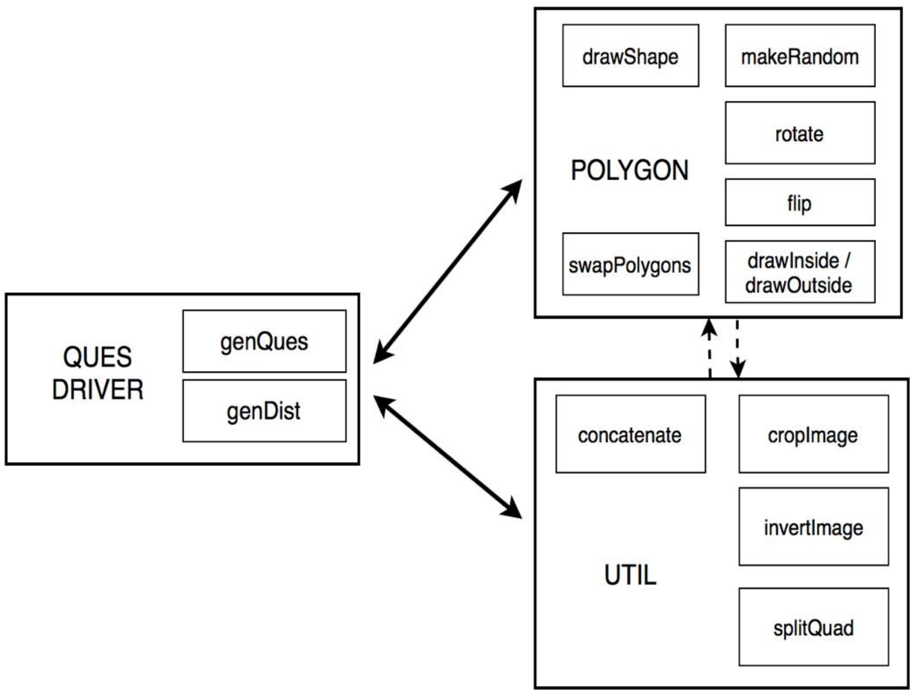


FIG 5.1 Architecture Diagram for Non-Verbal Question Generation

A Python class, Polygon, provides a wrapper and methods to draw and manipulate basic polygons shapes. It also handles creating arrows, circles, and different Unicode characters. The Util class, provides utility methods to do image manipulation to generate the final questions. A Question Driver, is a Python script that implements logic for generating questions and distractors for a particular type of question. It utilizes the generic framework and uses methods provided by Polygon and Util, in a modular way.

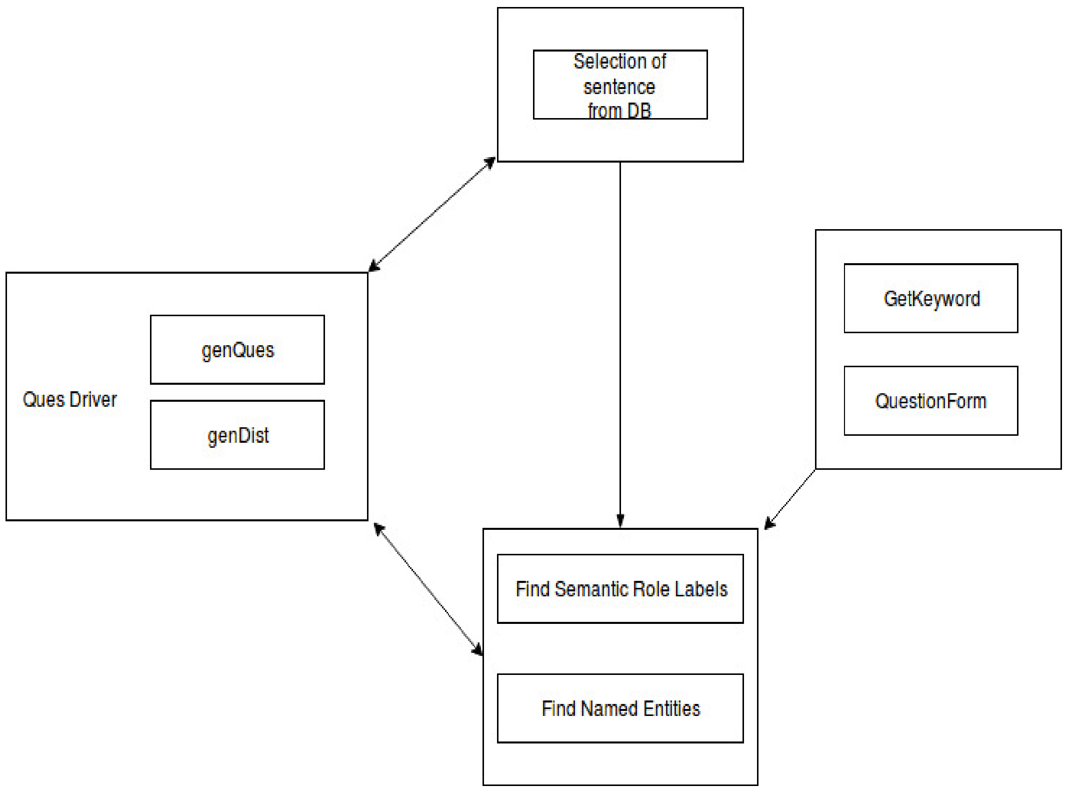


FIG 5.2 Architecture Diagram for MCQ Generation

A python class doc2vec provides methods for getting a sentence from database, finding similar sentences to that and generating the question. We created a database of sentences which have more information, and information which can be used to in generating the questions. The sentence is selected from the database and the passed to the component which finds the semantic labels and named entities from the sentence. These labels are then used to decide the key and the distractors from similar sentences from the database. Then these keywords and the sentence are passed to the question generation module which puts all these information in the question form.

5.2 Sequence Diagram

5.2.1 Non-Verbal Question

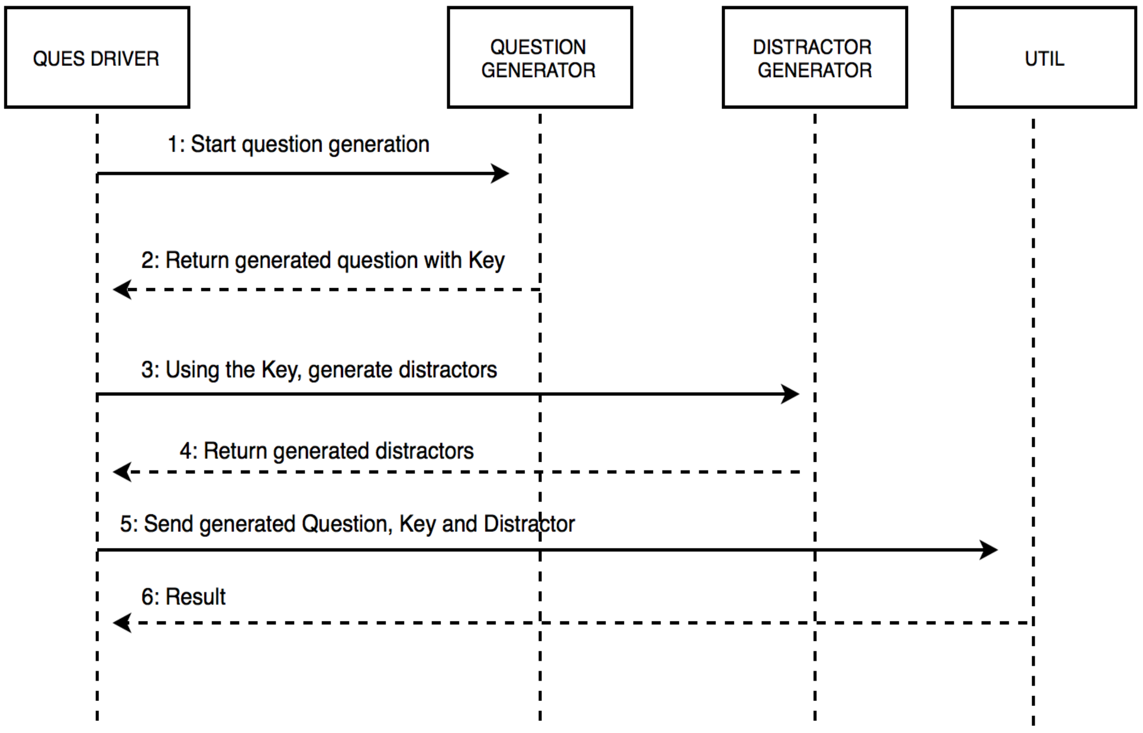

QUES DRIVER
QUESTION GENERATOR
DISTRACTOR GENERATOR
UTIL
1: Start question generation
2: Return generated question with Key
3: Using the Key, generate distractors
4: Return generated distractors
5: Send generated Question, Key and Distractor
6: Result

FIG 5.3 Sequence Diagram for Non-Verbal Question Generation

For Non-Verbal Question Generation, the first step is to generate a randomized question. The framework creates a new question and a key based on the type of question selected. Step 3 is to generate good distractors from the question and the given key. Once the distractors are generated, the UTIL component concatenates the final distractor images with the Key and generates a visibly suitable question

5.2.2 Multiple Choice Question Generation

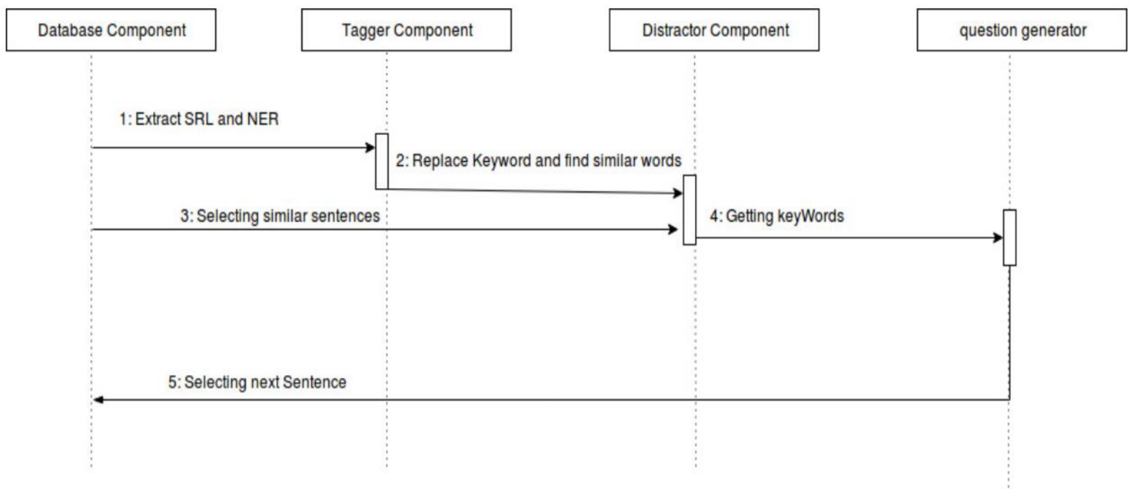


FIG 5.4 Sequence Diagram for MCQ Generation

For MCQ generation, Step 1 is to select a question statement at random from a database of sentences. After NER and SLR, the key from the sentence and the distractors from similar sentences are identified.

CHAPTER 6

IMPLEMENTATION

The main task was to automate question generation for a particular domain. This introduces problems of generating questions at scale. Producing questions at scale can only be done if the framework is robust enough. The robustness can be ensured by keeping the process generic with randomized components

6.1 Non Verbal Question Generation

6.1.1 Algorithm

1. Generate a primitive shape.
2. If composite figure is generated, then go to step 3. Else go to step 1.
3. Randomly choose transformations for the composite figure.
4. Repeatedly apply the transformations in order to obtain a sequence of images.
5. Choose a similar transformation on the same composite figure to generate distractors

6.1.2 Class Diagram

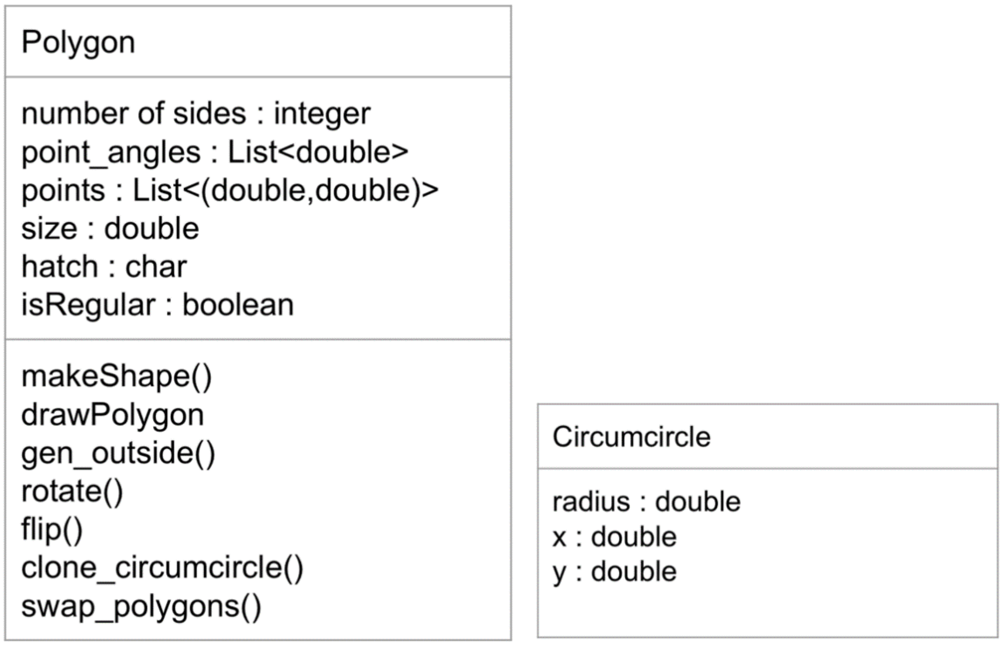


FIG 6.1 Class Diagram

6.1.3 Data Flow Diagram

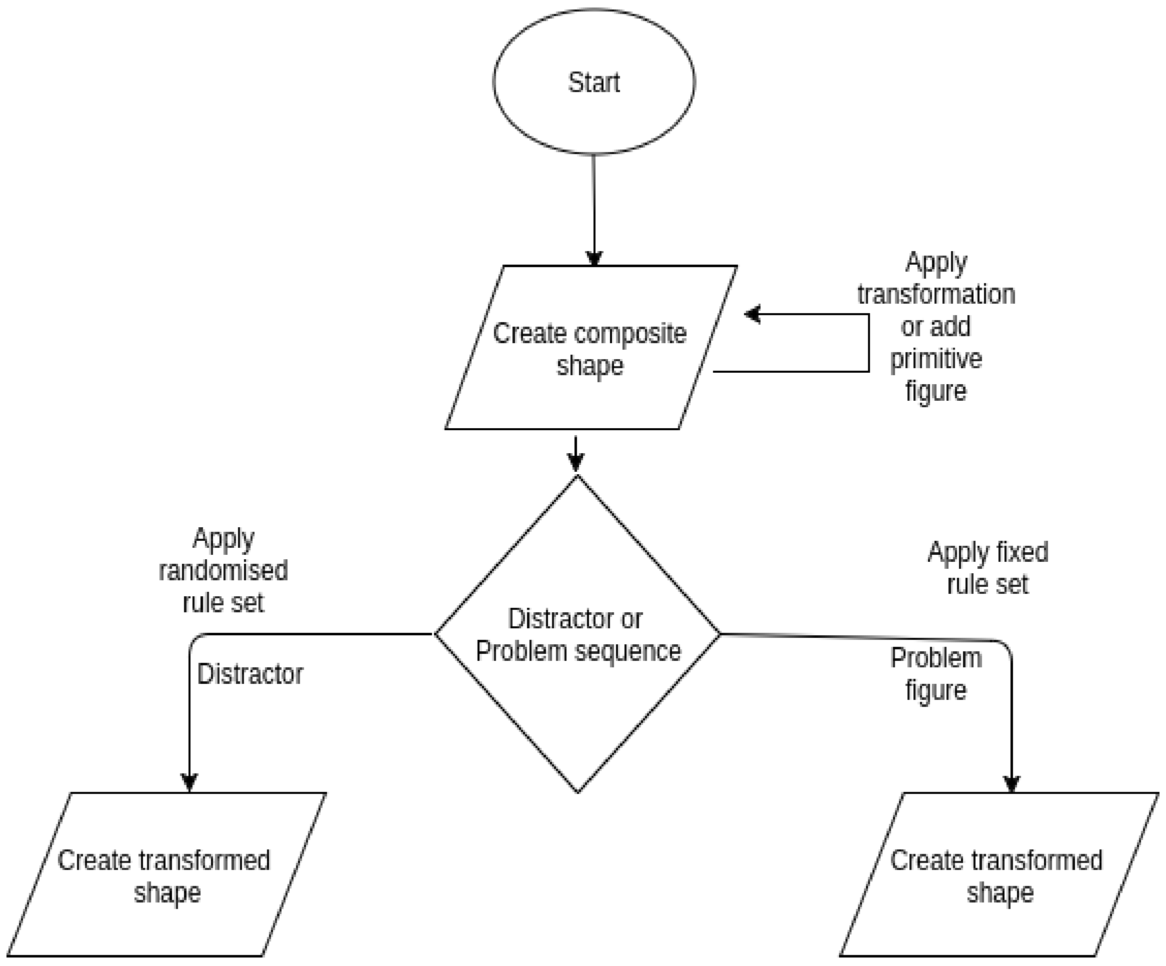


FIG 6.2 Data Flow Diagram – Non-Verbal Question Generation

The initial step in creating the diagram is to generate primitive shapes and arrange them with some rules to get a COMPOSITE shape. Then transformation or addition or deduction of figure is done on the original composite shape obtained. Now depending upon the type of question to be generated we apply some fixed set of rules on the original composite shape.

To generate distractors for the problem we generate composite figures similar to the original composite figure obtained in the initial step. Then we apply some random set of rules on the figures generated to get the distractors. The correct answer and the distractors are put in the random order, so there is no pattern in providing the options.

6.2 Multiple Choice Question Generation

6.2.1 Algorithm

1. Build the Knowledge Base by extracting the sentences which have the semantic attributes from the dataset.
2. Select a question sentence and identify the semantic type of the keyword by parsing it semantically.
3. For each question sentence: Measure the similarity between the question sentence and all sentences in the knowledge base. Sort the obtained similarity values.
4. Return the three sentences that have the highest similarity values.
5. Return three keywords of the three sentences as distractors and identify their types.

6.2.2 Data Flow Diagram

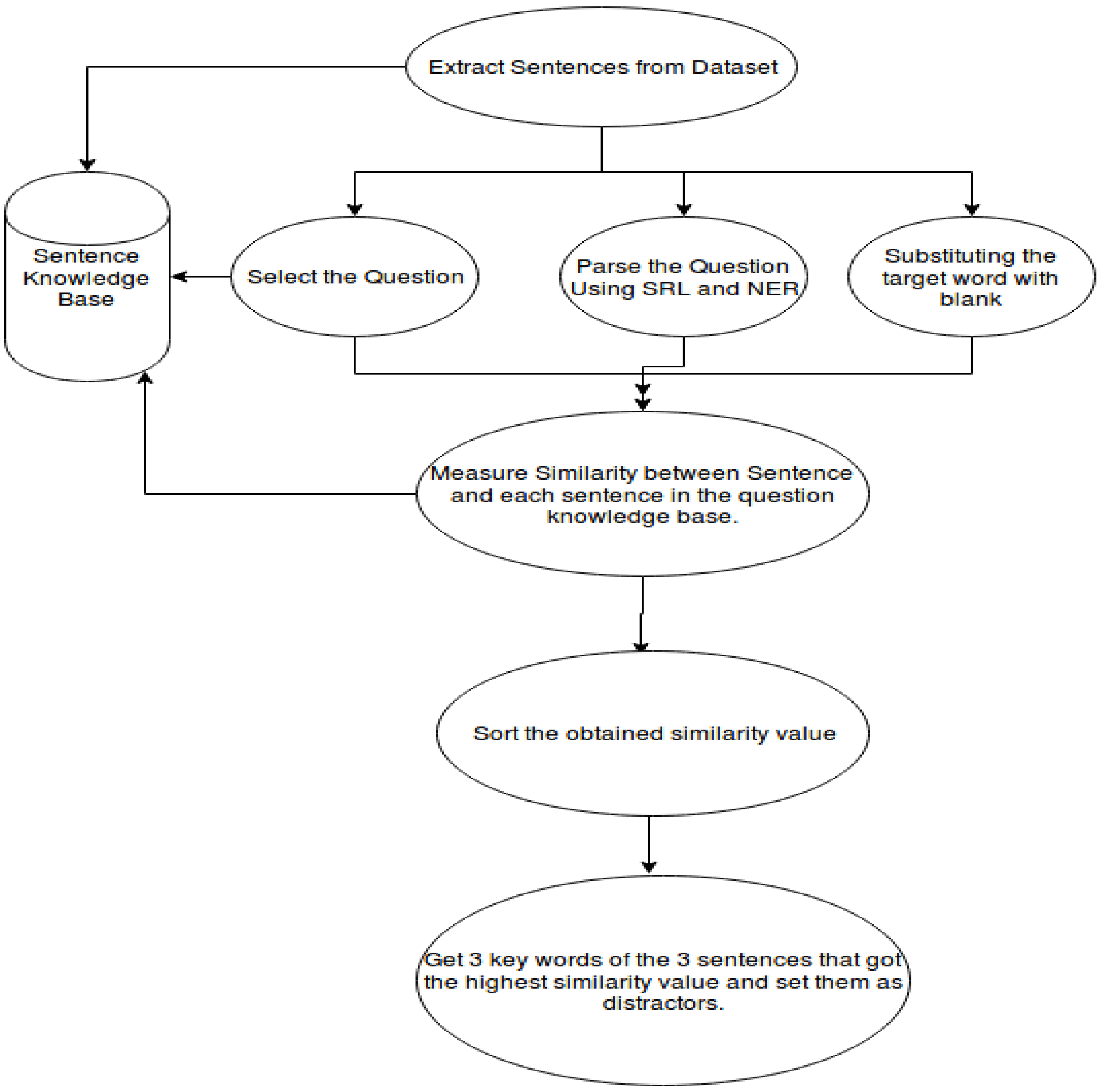
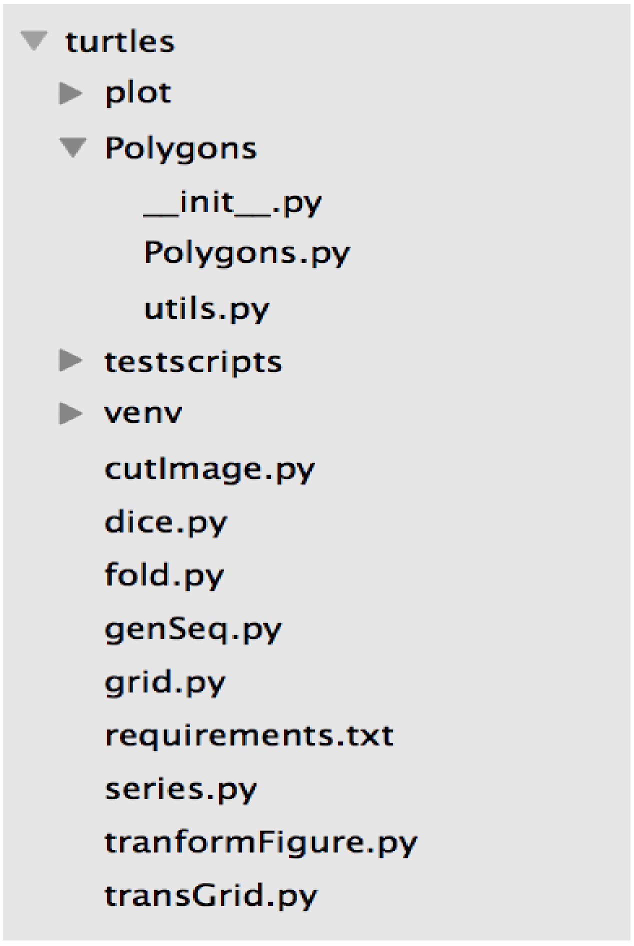

Extract Sentences from Dataset
Sentence Knowledge Base
Select the Question
Parse the Question Using SRL and NER
Substituting the target word with blank
Measure Similarity between Sentence and each sentence in the question knowledge base.
Sort the obtained similarity value
Get 3 key words of the 3 sentences that got the highest similarity value and set them as distractors.

FIG 6.3 Data Flow Diagram - MCQ Generation

For generating MCQ's on text basis, initial step is to construct the sentence knowledge database which contains the sentences which are rich semantically. After creating database a sentence is picked up from database then SRL and NER tags are found in the given sentence by parsing. We find the keyword based on the tags and replace the keyword with a blank. Similar sentences are found out using the string matching and character matching algorithms. We sort the obtained sentences on the basis of similarity values. After finding the top 3 sentences, we parse these top sentences and get SRL and NER tags for these sentences. We find the keywords for these sentences and pass it to the module which frames the question.

6.3 Codebase Structure



```
▼ turtles
    ▶ plot
    ▼ Polygons
          __init__.py
          Polygons.py
          utils.py
    ▶ testscripts
    ▶ venv
      cutImage.py
      dice.py
      fold.py
      genSeq.py
      grid.py
      requirements.txt
      series.py
      tranformFigure.py
      transGrid.py
```

FIG 6.4 Code Layout

The main component of the framework is the **Polygon** class found in Polygon.py. Utils.py contains the utility methods that we need for image manipulations. **plot** directory consists of the questions that would be generated. All other python files, represent a driver script representing a particular type of question to be generated. These files contain the logic pertaining to a particular type of question and the use the methods provided by Polygons and Utils component. **venv** is the virtual environment setup for Python development.

6.4 Sample Code

```
class Polygon :

    @staticmethod
    def getHatches():
        return ['-', '+', 'x', '\\', '*', 'o', 'O', '.','/','|']

    def __init__(self, no_of_sides=4,size=30,isRegular=True,hatch=None,circumcircle=None):
        # size is for the circumcircle radius
        print("In polygons, size is ",size)
        self.size = size
        self.N = no_of_sides
        if isRegular == 'any':
            self.isRegular = random.choice([True, False])
        else:
            self.isRegular = isRegular
        self.points = []

        self.circumcircle = circumcircle

        if hatch == 'random':
            self.hatch = hatches[int(random.random()*len(hatches))]
        else:
            self.hatch = hatch


        # for alphabet
        # in deg
        self.alphabet_rotation = 0
        self.alphabet = None

        # The angles at which each point was drawn
        # This will be helpful in case of rotating and other things(I can't think of right now.)
        self.point_angles = []

    """
        This shape is drawn randomly
        considering this as the first shape to be drawn

        makeRandomCircumcircle: makes a Circumcircle with random Center
    '''
    def makeRandomCircumcircle(self):
        # The shape is randomly drawn
        # Step1 : Make the circumcircle randomly

        # generate center points between 0 and 100
        self.circumcircle = Circumcircle(self.size, round(rnd(zeroToOne=1)*CANVAS_SIZE), round(rnd(zeroToOne=1) *
                                                                                              CANVAS_SIZE))
        # print "Random Circumcircle",self.circumcircle

        # Step2 :  Then call makeShape()
        self.makeShape()


    """"
    Given a Circumcircle and No_of_points generate points for the
    polygon
    """
    def makeShape(self):
        # Uses the information - circumcircle(radius,x,y) + no_of_sides + isRegular to generate vertices
        # for the polygon

        # Start with empty lists of points
        self.points = []
        self.point_angles = []

        # Pick a random angle
        start_angle = rndangle()

        if self.N == 0:
            # A circle
            # No need to generate points
            pass
        elif self.N == 1:
            # The center of the circumcircle is the point
            # No need to generate points
            # pass
            if self.alphabet == None:
                self.type='alpha'
                self.alphabet = random.choice(list(string.letters) + [u'\u2605',u'\u25DF',u'\u2020',u'\u002B'])
            pass
        elif self.N == 2:
            # An arrow
            # Two points that are opposite to each other
            self.point_angles = [start_angle, start_angle + math.pi]
            self.gen_points()
        elif self.isRegular:
            # print(self," is Regular")
            # Then theta is incremented uniformly by 360/N
            angle_increment = 2*math.pi / self.N

            # print("Angle of increment", angle_increment)
            self.points = []
            self.point_angles = []
            # Add points
            for i in range(self.N):
                # print(start_angle + i*angle_increment)
                cur_angle = start_angle + i*angle_increment
                self.point_angles.append(cur_angle)
                self.points.append(getpoints(self.circumcircle.radius, cur_angle, self.circumcircle.x, self.circumcircle.y))

        else:
            # Then theta is incremented by randangle
            # NOTE: A minimum value should be defined for increment,
            # otherwise the generated points would be too close to
            # each other
            # NOTE: We need to be sure that all points are unique!!

            # Currently increment with one of the values in angle_increment
            # angle_increment = [math.pi / 8, 2*math.pi / 6 ]

            self.points = []
            self.point_angles = []

            angle_increment = 2*math.pi / self.N
            # Add points
            for i in range(self.N):
                # print(start_angle + i*angle_increment)
                cur_angle = start_angle + i*angle_increment
                # Now modify the genreted angles a little bit
                # We increment/decrement angle by any number between 0 and 30 degrees
                cur_angle  += random.random() * IRREGULAR_ANGLE * ( -1 if random.random() < 0.5 else 1 )
                self.point_angles.append(cur_angle)

            self.gen_points()

        # point_angles can be any possible angles between [start_angle , start_angle + 2*pi].
        # it's better to convert them between 0 and 2*math.pi
        # To standardize the point angles generated, subtract 2*pi from any angle that was generated
        # if it is greated than 2*pi and then sort (sorting is not required but we just  want to keep it in a way that
        # all angles are in increasing order between 0 and 2*pi)
        # print ("Before ",self.point_angles)
        for i in range(len(self.point_angles)):
            if self.point_angles[i] > 2*math.pi:
                self.point_angles[i] -= 2*math.pi
        # print ("After ",self.point_angles)


    def setSize(self,size=10):
        self.circumcircle.radius = size

    def setHatch(self,hatch=''):
        self.hatch = hatch

    def isCircle(self):
        return self.N == 0 or self.N > CIRCLE_LIMIT_POINT

    def drawPolygon(self):
        points = self.points

        if self.N == 0:
            self.drawCircle()
        elif self.N == 1:

            # http://matplotlib.org/users/text_props.html
            # \u2726.\u2727,\u066D
            # normal star = \u066D
            # https://en.wikipedia.org/wiki/Star_(glyph)
            # https://en.wikipedia.org/wiki/List_of_Unicode_characters

            # BIG STAR - \u2605
            #  arc- \u25DF
            # Jesus cross - \u2020
            # plus - \u002B

            plt.gca().text(self.circumcircle.x, self.circumcircle.y, self.alphabet,
                    # rotation value should in degrees
                    rotation=self.alphabet_rotation * (180/math.pi) ,
                    fontsize=self.size, color='black',
                    multialignment='center',
                    verticalalignment='center', horizontalalignment='center',
                    # fontproperties=zhfont1
                )

        elif self.N == 2:
            # self.drawCircle()
            self.drawArrow()
        else:
            # FOR POLYGONS
            # fill = True if random.random() < 0.5 else False
            fill=False
            if not self.hatch:
                # don't do anything
                hatch = None
            elif self.hatch == 'random':
                # pick a random hatch
                hatch = hatches[int(random.random()*len(hatches))]
                self.hatch = hatch
            else:
                hatch = self.hatch

            polygon = plt.Polygon(points,fill=fill,hatch=hatch)
            plt.gca().add_patch(polygon)

    def drawCircle(self):
        # fc='y'

        if not self.hatch:
            # don't do anything
            hatch = None
        elif self.hatch == 'random':
            # pick a random hatch
            hatch = hatches[int(random.random()*len(hatches))]
            self.hatch = hatch
        else:
            hatch = self.hatch

        circle = plt.Circle((self.circumcircle.x, self.circumcircle.y), self.circumcircle.radius, fill= False, hatch=hatch)
        plt.gca().add_patch(circle)

    def drawArrow(self):
        # arrow = plt.arrow(self.points[0][0], self.points[0][1], self.points[1][0]-self.points[0][0],self.points[1][1]-self.points[0][1],fc="k", ec="k",head_width=0.05, head_length=0.1)
        # arrow = plt.arrow( 0.5, 0.8, 0.0, -0.2, fc="k", ec="k",head_width=0.05, head_length=0.1 )
        # arrow= plt.arrow(0, 0, 0.5, 0.5, head_width=0.05, head_length=0.1, fc='k', ec='k')
        print self.points[0][0], self.points[0][1], self.points[1][0]-self.points[0][0],self.points[1][1] - self.points[0][1]
        x1, y1 = self.points[0]
        x2, y2 = self.points[1]
        dx,dy = x2-x1, y2-y1

        # FIX make head_width and head_length relative to something so that it scales.
        plt.arrow(x1,y1,dx,dy, head_width=abs(min(dx, dy) * 0.1), head_length=abs(min(dx, dy) * 0.1), fc='k', ec='k' )

        # Works without gca shit. Don;t know why :)
        # plt.gca().add_patch(arrow)
        pass

    """
        Generate this object outside the poly
    '''
    def gen_outside(self,otherpoly):
        # otherpoly should be a polygon
        if not type(otherpoly) == type(self):
            # raise error
            print("Something is wrong: Wrong type"+type(self))
        else:

            total_distance  = otherpoly.circumcircle.radius + self.size # +some_random_distance

            # Get a random angle to draw the new image
            draw_angle = rndangle()
            self.gen_point_angles()
            self_center = getpoints( total_distance, draw_angle, otherpoly.circumcircle.x, otherpoly.circumcircle.y )
            self.circumcircle = Circumcircle( self.size, self_center[0], self_center[1])
            self.gen_points()

            # Genereate points for otherpoly
            # self.makeShape()

    def move_outside(self,otherpoly):
        pass

    """
        Generate this object outside the poly
    '''
    def gen_outside_all(self, *otherpolys):

        for i in otherpolys:
            assert (type(i) == type(self))

        common_center_x = sum([poly.circumcircle.x for poly in otherpolys]) / len(otherpolys)
        common_center_y = sum([poly.circumcircle.y for poly in otherpolys]) / len(otherpolys)
        distance = self.size + sum([poly.size for poly in otherpolys])

        # Get a random angle to draw the new image
        draw_angle = rndangle()

        self_center = getpoints( distance, draw_angle, common_center_x, common_center_y )
        self.circumcircle = Circumcircle( self.size, self_center[0], self_center[1])

        # Genereate points for otherpoly
        self.makeShape()

    """
        Generate this object inside the otherpoly
    '''
    def gen_inside(self,otherpoly):
        self.circumcircle = Circumcircle(otherpoly.circumcircle.radius / 2 , otherpoly.circumcircle.x, otherpoly.circumcircle.y )
        self.makeShape()

    def move_inside(self,otherpoly):
        pass

    def rotate(self,theta=math.pi/2):
        if self.N == 1:
            # Alphabet rotation should be in degrees
            # theta here is in rads as for everything else
            self.alphabet_rotation += theta

        elif not self.isCircle():
            # rotate the current polygon clockwise by theta
            self.points = []
            print(len(self.point_angles))
            # Find new points drawn at new angles.
            for cur_angle in self.point_angles:
                self.points.append(getpoints(self.circumcircle.radius, cur_angle + theta,
                                             self.circumcircle.x, self.circumcircle.y))

            # print("New length",len(self.points))
            # Update the angles ate which vertices are drawn
            self.point_angles = [ cur_angle + theta for cur_angle in self.point_angles]
            pass

    def add_vertex(self):
        if not self.isCircle():
            # Add a random vertex to the figure
            self.N += 1
            self.makeShape()

    def delete_vertex(self):
        if not self.isCircle():
            if self.N <= 3:
                # Can't do nothing
                raise "Error: Polygon can't exist with less than 3 vertices "
                pass
            else:
                self.N -= 1
            self.makeShape()

    """
    Change the circumcircle for the Polygon
    """
    def clone_circumcircle(self,otherpoly):
        self.circumcircle = Circumcircle(otherpoly.circumcircle.radius,otherpoly.circumcircle.x,otherpoly.circumcircl
e.y)
        self.makeShape()

    """
    Flip the image
    how : vert - about vertical axis
          hori - about horizontal axis
    """
    def flip(self,how='vert'):
        if self.N == 1:
            # Alphabet
            if how =='vert':
                self.alphabet_rotation='vertical'
            else:
                self.alphabet_rotation = 'horizontal'

        elif not self.isCircle():
            pts = []
            if how == 'vert':
                for pt in self.points:
                    x,y = pt
                    xdist = self.circumcircle.x - x

                    x,y = 2 * self.circumcircle.x - x, y
                    pts.append((x,y))
            elif how == 'hori':
                print "hori"
                for pt in self.points:
                    x,y = pt
                    x,y = x, 2* self.circumcircle.y - y
                    pts.append((x,y))

            # Update the points
            self.points = pts

            # Points updated so update the angles as well
            self.gen_point_angles()

    def gen_point_angles(self):
        # Given the points are generated, find the angles in which they were
        # generated.
        if not self.isCircle():
            # x = r cos(theta)
            # y = r sin(theta)
            point_angles = []
            for pt in self.points:
                #returns an angle between -
pi to +pi  ; since the y and x values are known, atan2 method can find the quadrant where
                # the angle is drawn.
                x, y = pt
                angle = math.atan2(y - self.circumcircle.y, x - self.circumcircle.x)
                if angle < 0:
                    angle = (2 * math.pi) + angle
                point_angles.append(angle)

            # point_angles = sorted(point_angles)

            # print("Earlier",self.point_angles)
            self.point_angles = point_angles
            # print("Now",self.point_angles)


    def get_point_angles(self):

        self.gen_point_angles()
        return self.point_angles

    def gen_points(self):
        if not self.isCircle():
            # Given point angles and the circumcircle, draw
            self.points = []
            for angle in self.point_angles:
                self.points.append(getpoints(self.circumcircle.radius, angle,
                                             self.circumcircle.x, self.circumcircle.y))

    def swap_polygons(self,otherpoly):
        # swap the circumcircles of the two polygons and then replicated points
        # at the sme angles and then draw it.

        # Making sure that the point angles are properly calculated and preserved
        self.gen_point_angles()
        otherpoly.gen_point_angles()

        # Swap the circumcircles
        self.circumcircle, otherpoly.circumcircle = otherpoly.circumcircle, self.circumcircle

        # Once the circumcircles are swapped, re-make the points.
        self.gen_points()
        Otherpoly.gen_points()
```

FIG 6.5 Code Sample for Polygon Class

CHAPTER 7

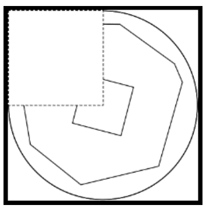
RESULTS AND DISCUSSION

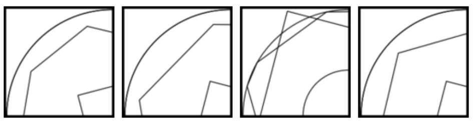
In this section we discuss the type of questions we could generate.

7.1 Types of questions

7.1.1 Cut

Ques: Identify the figure that completes the figure.

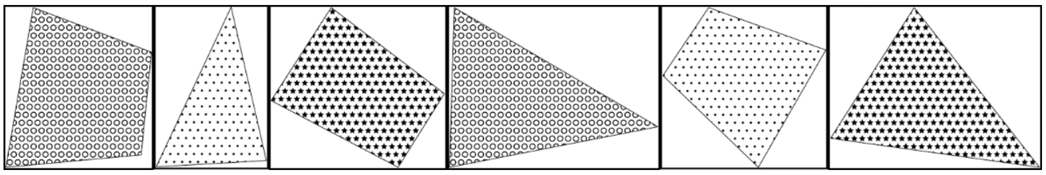


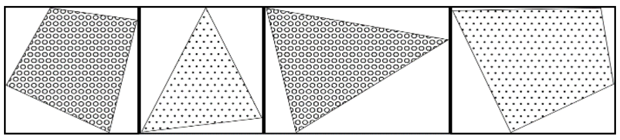
Options: 

In this type of question, a figure is given and a part of it is cut out. The user has to guess which of the options completes the figure. The limitation of these questions is that the question figure is made up of concentric shapes. This was done to ensure that the pattern is consistent along the four quadrants

7.1.2 Series

Ques: Guess the next figure in the series.

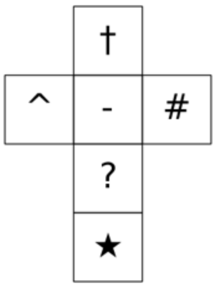


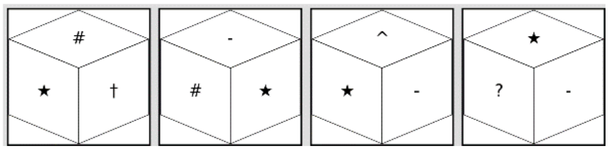
Options: 

Here, a series of figures is given which follow a certain pattern of transformation. The user here needs to guess which of the options the next figure in the series is. The limitation with these questions was that we were not able to create composite figures consisting of more than a certain number of primitive shapes as this usually led to ambiguity of the question.

7.1.3 Dice

Ques: Choose the box that is made out of the given piece of paper.

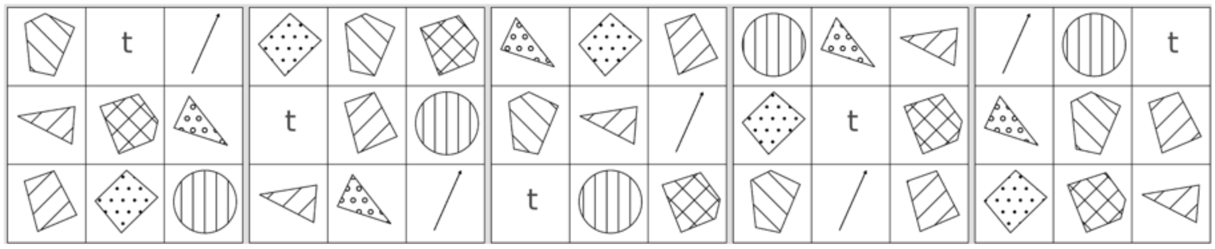


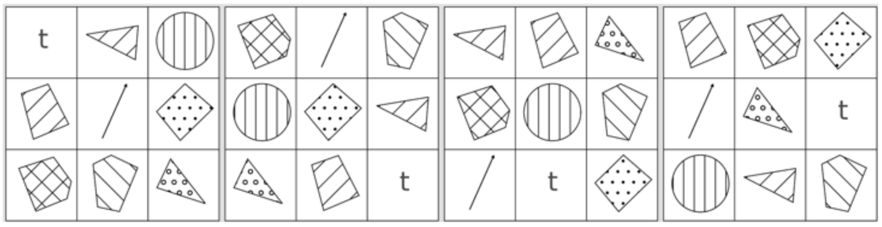
Options: 

In a dice problem, an open cube figure is given and the user has to choose which of the options best describes a cube made out of the given paper piece.

7.1.4 Figure Matrix and Sequence

Ques: Select the figure which will continue the same series as the given 5 images.

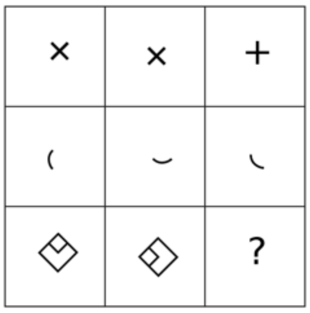


Options: 

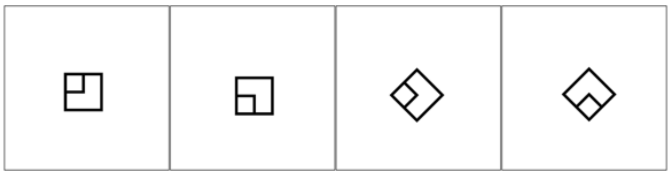
This question type is a combination of figure matrix and sequence question. Every individual figure in a block of the matrix can have any transformation or primitive rule attached to it. Also, every block is following a certain pattern while moving inside the matrix. With this type of question we were able to generate an optimum level of complexity while maintaining a certain clarity in representation.

7.1.5 Grid

Ques: Which option replaces the question mark?

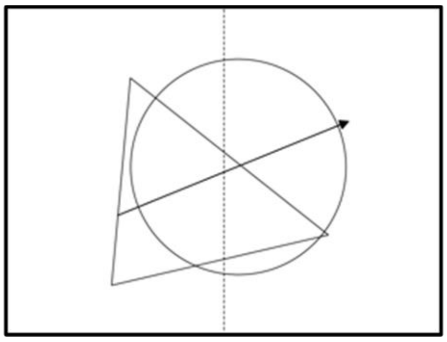


Options:

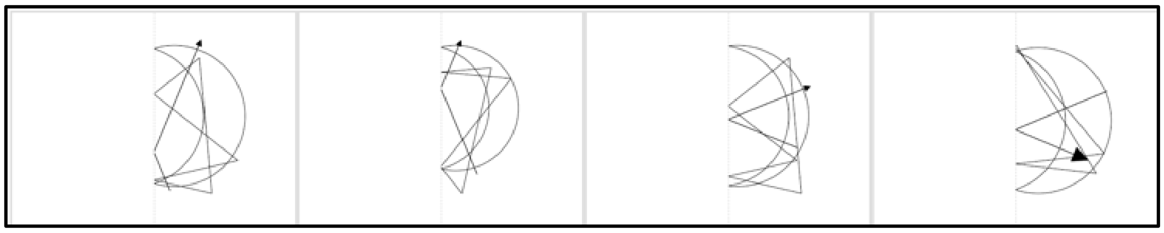


7.1.6 Fold

Ques: Find out amongst the four alternatives as to how the pattern would appear when the paper is folded on the dotted line.

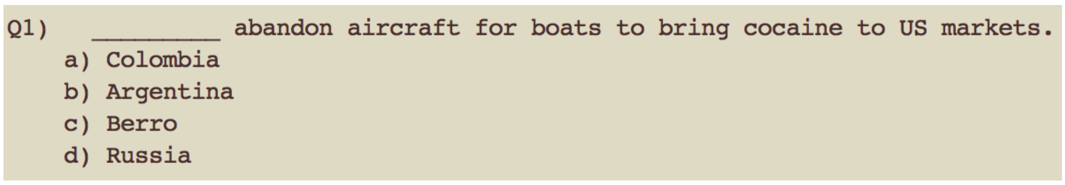


Options:

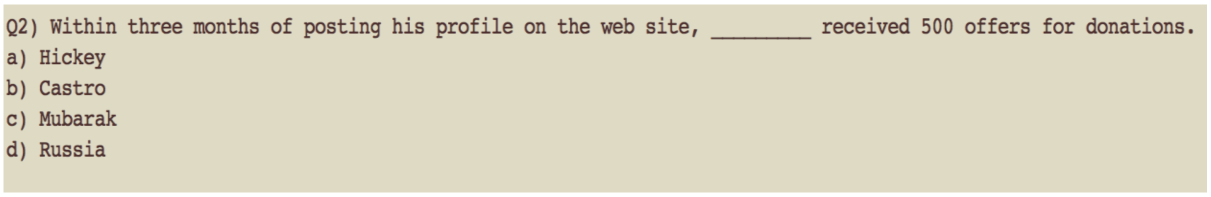


7.1.7 Multiple Choice Questions from text

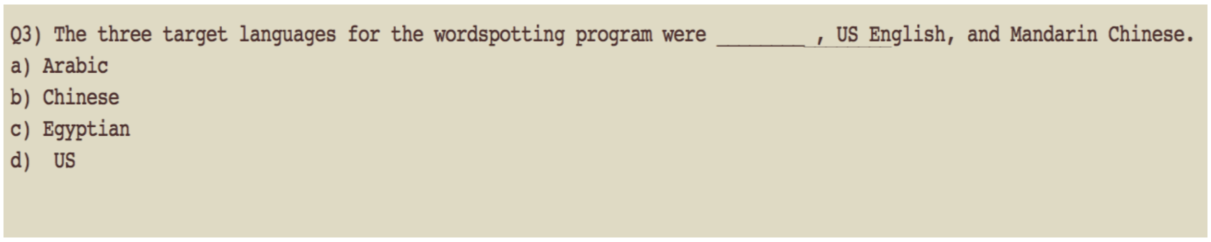
Some of the sample questions that our systems could generate. The correct word has been replaced by a blank.



```
Q1)   _________ abandon aircraft for boats to bring cocaine to US markets.
    a) Colombia
    b) Argentina
    c) Berro
    d) Russia
```



```
Q2) Within three months of posting his profile on the web site, ________ received 500 offers for donations.
a) Hickey
b) Castro
c) Mubarak
d) Russia
```



```
Q3) The three target languages for the wordspotting program were ________ , US English, and Mandarin Chinese.
a) Arabic
b) Chinese
c) Egyptian
d)  US
```

7.2 Future Enhancements

- The framework to generate non-verbal questions could include free-flow diagrams or some way of incorporating symmetrically drawn shapes and figures that would make the generated questions resemble that of a human generated one.

- The framework to generate MCQs could expand to more categories and use more well linked databases to generate more interesting questions.

CHAPTER 8

RETROSPECTIVE

8.1 What went well?

We were working in a domain which is totally unexplored as per our knowledge and it was an enriching experience to have accomplished the fundamental goal of the project. We were able to generate questions of various types which are very close to the questions made manually for MOOCs. We were also able to induce a certain difficulty level in the questions which was very crucial for the project to be practically useful. We got the opportunity to learn Prolog and the application of matplotlib. We also learnt how to pick ideas from different domains and try to use them to formulate a solution for our target domain.

8.2 What did not go well?

We always knew that one of the most difficult task in this project would be to cover all the types of questions. We also had to accomplish this while keeping the framework as generic as possible. This made it difficult for us to incorporate images or free flow figures. This restricted our variety a lot.

8.3 Learnings beyond the technologies

Time management is one of the best lessons we learnt from this project. We realized the importance of the previous work carried out in different domains. We learnt how to work on the project as a team while being virtually present.

This project was intrinsically different from other projects we had worked on earlier because we had to come with our own problem statement and had to revise it after every iteration of research and discussions with our guide.

BIBLIOGRAPHY

[1] Singhal, R., Henz, M. and Goyal, S., 2015, June. **A framework for automated generation of questions based on first-order logic**. In *International Conference on Artificial Intelligence in Education* (pp. 776-780). Springer International Publishing

[2] Singhal, R., Henz, M. and McGee, K., 2014. **Automated Generation of High School Geometric Questions Involving Implicit Construction**. *In CSEDU* (1) (pp. 467-472).

[3] Brown, J., Firshkoff, G. And Eskenazi, M. (2005**) Automatic Question Generation For Vocabulary Assessment**. *Proceedings Of Hlt/Emnlp*, 819–826. Vancuver, Canada

[4] Sung, L., Lin, Y., And Chern, M.(2007). **An Automatic Quiz Generation System For English Text**. *Seventh Ieee International Conference On Advanced Learning Technologies*

[5] Agarwal , M. And Mannem ,P. (2011). **Automatic Gap-Fill Question Generation From Text Books**. In *Proceedings Of The 6th Workshop On Innovative Use Of Nlp For Building Educational Applications*. Portland, Or, Usa. Pages 56-64

USER MANUAL

Prerequisites

1. Python
2. Matplotlib
3. ImageMagick
4. Git

Setup

1. Clone the repository from Github.

```
git clone https://www.github.com/GiriB
```

2. Install requirements.

```
pip install –r requirements.txt
```

Running

Go to the project root directory and run any file to generate question for a particular type.

```
python cutImage.py
```