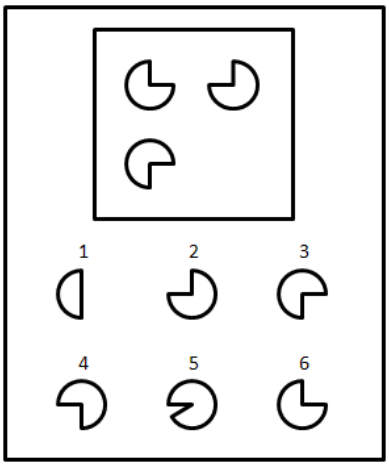
# Knowledge Based AI – Visual Reasoning

**Vinay Bettadapura**

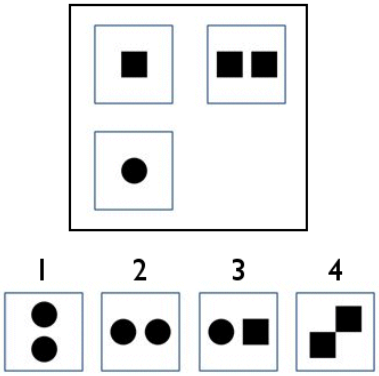
## 1. Introduction

The goal of the project is to learn about visual representations and reasoning. The task here is to address three of the 2x2 Raven's matrix problems. However, unlike projects 1 and 2, this time the input is in the form of an image. The problems considered in this assignment are:

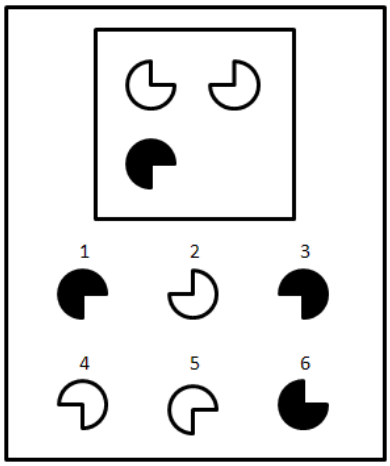
**Problem 1:**




**Problem 2:**




**Problem 3:**




In projects 1 and 2, we came up with propositional representations for the problems and gave that as the input to our program. In this project, we have two possible ways by which we can solve the problems using visual reasoning:

1. The first approach has 2 steps: In the first step, automatically extract propositional representations from the problems using image-processing/computer-vision techniques. In the second step, give this propositional representation as input to the program that we have written for projects 1 and 2. In this approach, we are automating the task of extracting the propositional representation (something that we had done manually in the previous projects).

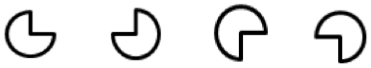
2. In the second approach, we work directly with the raw pixel data. We apply image-processing/computer-vision techniques to the images and directly infer the transformations (without going for an intermediate propositional layer). This is the approach taken in this assignment.

## 2. Shapes and Transformations

Before we look at the different possible transformations between the shapes, let us look at the various shapes that we have.

### 2.1. Shapes

The various shapes that the figures have (across all the three problems) are shown in the table below. The first column contains the shape names (some of the shape names are my own creation) and the second column shows the corresponding shapes:

| **Shape** | **Examples** |
|---|---|
| Semicircle |  |
| Three-Quarter |  |
| Almost-Circle |  |
| Circle |  |
| Square |  |

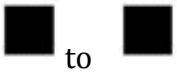
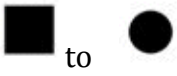
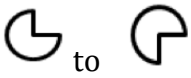
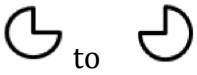
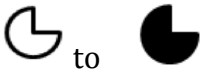
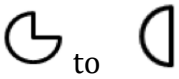
### 2.2. Fill-Color

The figures can have any one of the following fill-colors:

| **Fill-Color** | **Example** |
|---|---|
| White |  |
| Black |  |

## 2.3. Transformations

Let us now look at the different transformations that are defined in the system. Given any two images, the system checks for the following transformations between them:

| Transformation | Example |
|---|---|
| Same |  |
| Almost-Same<br>(Many of the pixels are present in both) |  |
| Horizontal-Replication |  |
| Horizontal-Flip |  |
| Vertical-Flip |  |
| Color-Change |  |
| Different |  |

## 3. System Architecture

A high level view of the system architecture is given below.

## 3.1. Stage 1 - Find the Transformation Between Two Figures

Figure 3.1 shows the Stage 1 of the architecture. In this stage, the 2 given figures (referred to as Figure 1 and Figure 2 in the block diagram) are compared against each other and the transformation that relates the two figures is found.

The comparisons are done by a “Transformation Analyzer” (the big block on the left in the figure), which compares the two shapes and checks if they are the same or almost the same, if they have been horizontally replicated, horizontally flipped, vertically flipped or if the color has changed. An example is shown in Figure 3.2.

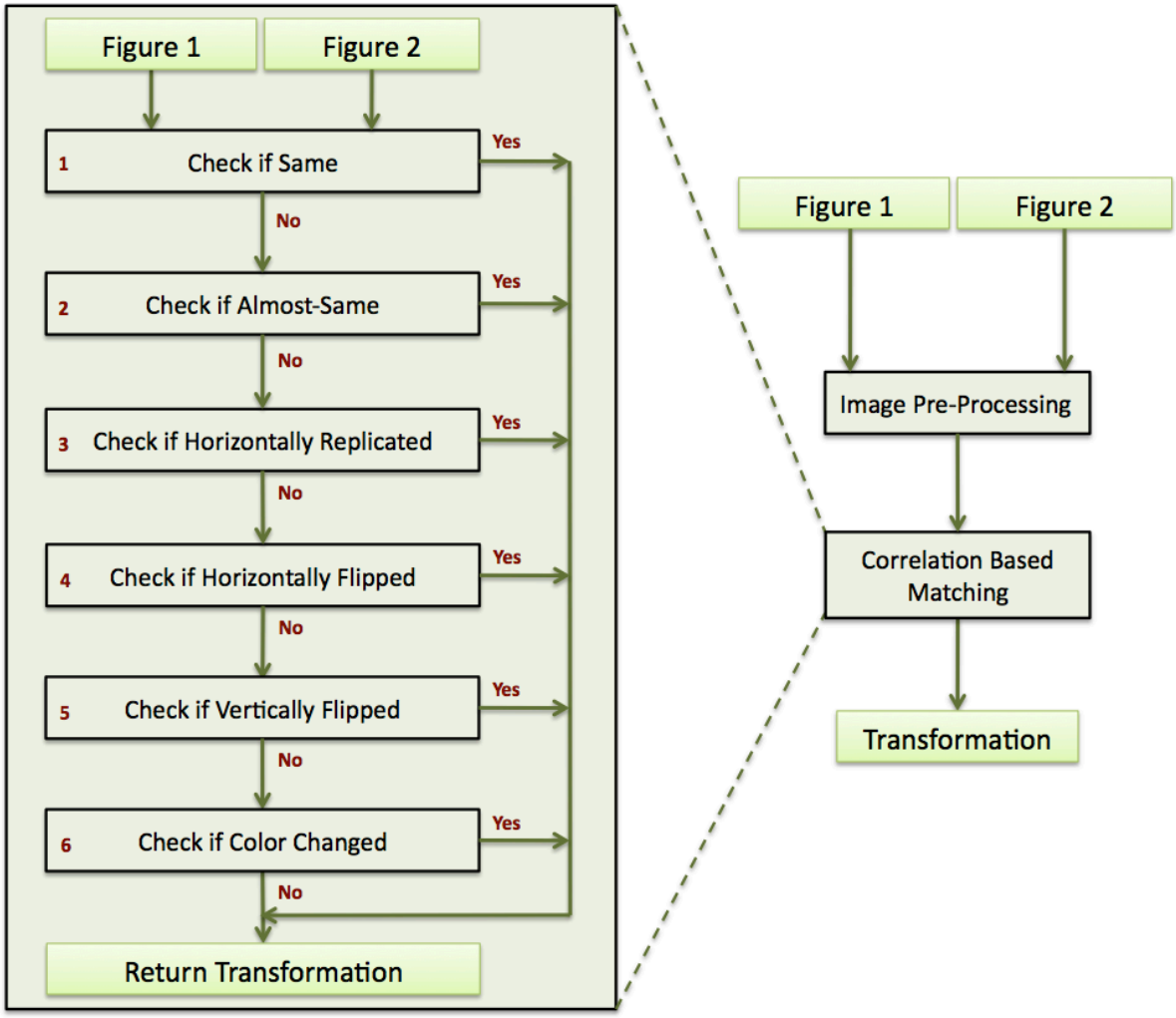



**Figure 3.1: Stage 1** - The 2 figures that have to be compared is given as the input. The two figures are compared for all possible transformations and the result is returned.

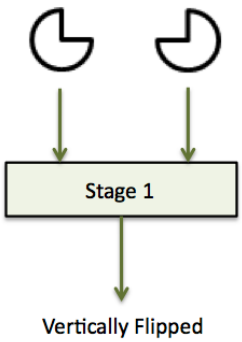



**Figure 3.2: Stage 1 example** - The 2 figures that have to be compared is given as the input. The two figures are compared and the transformation between them is returned.

Let us now look at how the transformations are found. At the heart of the system is a **Template Matching** algorithm that is widely used in the computer-vision community to match a given template against an image. Template matching is a technique for finding small parts of an image, which match a template image. For this assignment, I am making use of the "**OpenCV**" library for image manipulation and Template Matching.

A general example of Template Matching is shown in Figure 3.3:

**Figure 3.3:** An image of a cat is shown on the left. A template image (cat's head) is shown in the middle and the matched area denoted by a black bounding box is shown on the right.

A basic method of template matching uses a mask (the template), tailored to a specific feature of the search image, which we want to detect. The output will be highest at places where the image structure matches the mask structure, where large image values get multiplied by large mask values.

The template matching is performed using **Normalized 2D Cross-Correlation**. Let *I* be an image of size *(W x H)* and let *T* be a template of size *(w x h)* that has to be matched against the image. The normalized cross-correlation coefficient *R* is found using the following formula:

$$R(x,y) = \frac{\sum_{x',y'}(T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$$

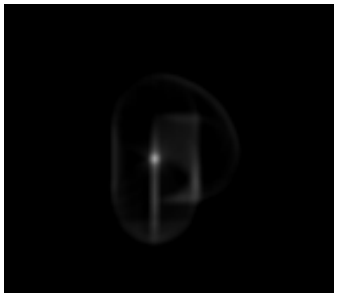
and

$$x' = 0...w-1, y' = 0...h-1$$

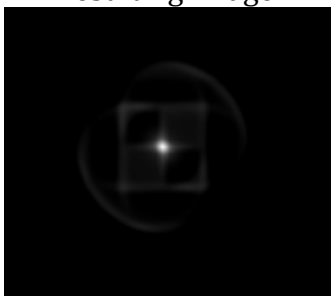
The resulting value *R(x,y)* is considered as the value in another image, which we will call as the Result-Image. Thus after we perform Normalized Cross-Correlation between the image and template, we get a result-image *R*. The pixel values of R are then scaled to the range [0, 255], where 0 denotes black and 255 denotes white. The point where we have a global maximum indicates the location of the best match. If the highest pixel value in the result-image is 255, then we have a perfect match, which indicates that the image and the template are identical. If the highest pixel value in the result-image is 0, then it indicates no match. Any value > 0 and < 255 gives us a score on the similarity.

Let us now look at some examples from the given problems. In the examples below, the image, the template, the result-image and the value of the highest pixel is shown:

**Normalized Cross-Correlation between different shapes:**

| Image | Template | Result-Image | Highest Pixel Value |
|---|---|---|---|
|  |  |  | **179** |

**Normalized Cross-Correlation between identical shapes:**

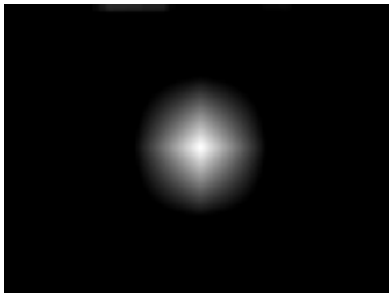
| Image | Template | Resulting Image | Highest Pixel Value |
|---|---|---|---|
|  |  |  | **255** |

As discussed above, we can see that we get a perfect match score of 255 when the template and the image are identical. For shapes that are different, we get a score that is less than 255.

Now, let us look at each of the transformations shown in Figure 3.1.

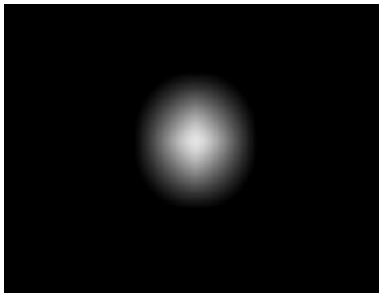
### 3.1.1. Check if they are "Same":

To check if the two images are same, we perform Normalized Cross-Correlation between the two images and then look at the highest pixel value in the result-image. If the value is greater than a threshold, we declare them to be "same". In the ideal case, we will get a value of 255 (indicating perfect match). But if the shapes are identical but are slightly modified versions of each other (ex: slightly scaled), then the value drops. For this assignment, I have set the threshold to be **235**. Any match that generates a value > 235, will be declared as "same". The threshold was determined empirically. Here is an example:

| Image | Template | Resulting Image | Highest Pixel Value |
|---|---|---|---|
|  |  |  | **255**<br><br>which is > 235<br><br>Thus, declared "same" |

### 3.1.2. Check if they are "Almost-Same":

When we are doing pixel-level comparisons, it is easy for the Normalized Cross-Correlation algorithm to get confused when two shapes share a lot of common pixels, but are not identical. An example is shown below. To differentiate such shapes from shapes that are exactly identical, we should get a value that is high but not high enough to qualify to be a perfect match. So any shapes that generate a result-image in which the highest pixel value is **> 220** but **< 235** will be determined to be "almost-same".

| Image | Template | Resulting Image | Highest Pixel Value |
|---|---|---|---|
|  |  |  | **229**<br><br>which is > 220 but < 235<br><br>Thus, declared "almost-same" |

### 3.1.3. Check if "Horizontally-Replicated":

Here we check to see if a figure has been replicated in the horizontal direction. Detecting horizontal-replication is a three-step process:

1. Detect sub-images in the given image.

2. Compare each sub-image against the template. Also, check for 3 things:
   a. Are there two sub-images?
   b. Are the sub-images same as the template?
   c. Are the sub-images located beside each other (horizontally)?

3. If all the three conditions in step 2 are true, declare the image to be a "horizontal-replicated" version of the template and exit. Else return false.

Consider the example below:

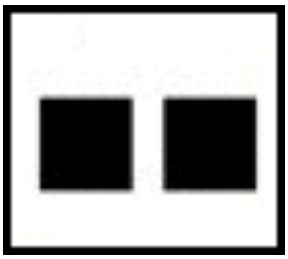

The first step is to detect the sub-images on the given image. This is called as "**Blob-Detection**" in the computer-vision community. Blob-detection invloves taking a binary image and labeling the connected components. For this, I am using the "**cvBlobsLib**" library which can be readily used with OpenCV.

Example: A binary image is shown below.

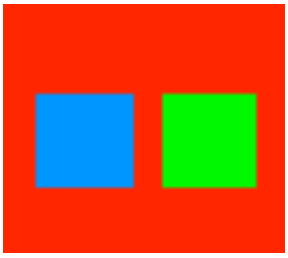
0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 1 1 1 0
0 1 1 1 0 0 1 1 1 0
0 1 1 1 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0

After we run the blob-detector, we get a set of different connected components. In the figure below, the different connected components are shown in different colors:


A real figure like this:



after blob-detection would become:



The next step has three sub-steps. In step 2a, we check to see if there are two sub-images. Before doing so, the biggest blob is discarded because it corresponds to the background. In the figure above, this would be the blob in red. Then we see that there are two blobs/sub-images remaining (the ones shown in blue and green).

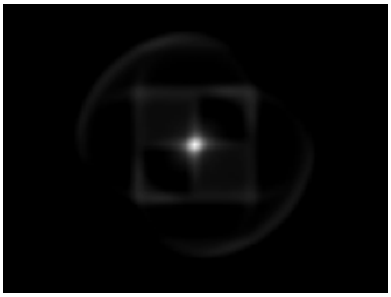
In step 2b, we compare each of the sub-images against the template. Each blob is considered as a regular image and as before Normalized Cross-Correlation is performed with the template. We check if both the subimages matches with the template image. For this, the same method described in section 3.1.1 is used.

The last sub-step is step 2c. In this step, we simply compare the center position of the two blobs against each other and check if they are beside each other on a horizontal axis.

Next we proceed to step 3 where we check if all the conditions of step 2 are satisfied. For the example that we have, all the conditions are indeed satisfied. Thus the image is declared to be a horizontally-replicated version of the template.

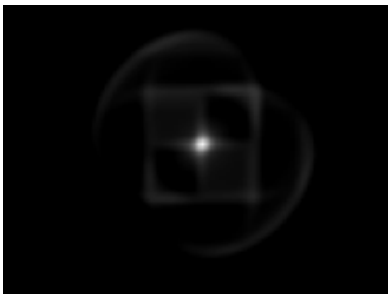
### 3.1.4. Check if "Horizontally-Flipped":

To check if the two images are "horizontally-flipped" versions of each other, the image is taken and flipped along the horizontal axis and then matched against the template using the same method described in section 3.1.1.

| Image | Horizontally Flipped Image | Template | Resulting Image | Highest Pixel Value |
|---|---|---|---|---|
|  |  |  |  | **251**<br><br>which is > 235<br><br>Thus, declared "same" and in turn "horizontally-flipped" |

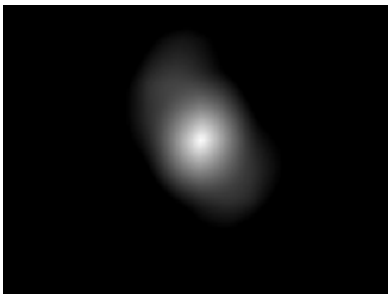
### 3.1.5. Check if "Vertically-Flipped":

To check if the two images are "vertically-flipped" versions of each other, the image is taken and flipped along the vertical axis and then matched against the template using the same method described in section 3.1.1.

| Image | Vertically Flipped Image | Template | Resulting Image | Highest Pixel Value |
|---|---|---|---|---|
|  |  |  |  | **246**<br><br>which is > 235<br><br>Thus, declared "same" and in turn "vertically-flipped" |

### 3.1.6. Check if "Color-Changed":

To check if the two images are "color-changed" versions of each other, the image is taken and filled with black-color and then matched against the template using the same method described in section 3.1.1. To color the image, I am using the OpenCV function **cvFloodFill**.

| Image | Colored Image | Template | Resulting Image | Highest Pixel Value |
|---|---|---|---|---|
|  |  |  |  | **253**<br><br>which is > 235<br><br>Thus, declared "same" and in turn "color-changed" |

If none of the 6 transformations are detected, then the image and the template are declared to be "Different".

## 3.2. Stage 2 – Performing Horizontal and Vertical Matches

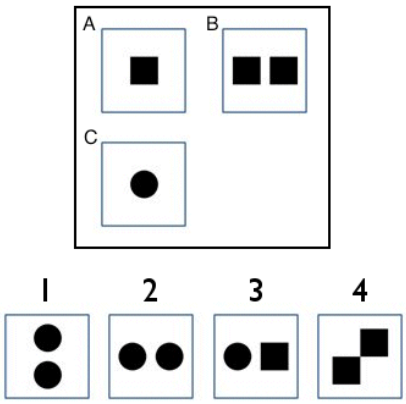
The 2x2 Raven's test problems have analogies in both the horizontal direction and in the vertical direction. Thus, we have to analyze the evidences both **horizontally** and **vertically**. To solve these kinds of problems, we run the problem through Stage 1 of our system in two passes:

1. In the first pass, the evidences in the first row are considered to be A and B. The transformation between them is learnt. Then the single evidence in the second row is considered to be C and is matched against all the choices and the best choices are obtained.

2. In the second pass, the evidences in the first column are considered to be A and B. The transformation between them is learnt. Then the single evidence in the second column is considered to be C and matched against all the choices and the best choices are obtained

The best choice is the one that gets the highest score in both the passes.

Performing both row-wise and column-wise comparisons is very important. To understand why, let us look at an example: Consider Problem 2:




If we perform only row-wise comparisons, we see that there are two ways of going from A to B. Either we can horizontally-replicate A to obtain B or we can take whatever shape is in A and place a square to the right of it to obtain B. Based on this understanding, we see that there are two correct answers. Choice 2 can be obtained by horizontally-replicating and Choice 3 can be obtained by placing a square to the right. So, we have no way of picking the right answer. But when we look at the problem column-wise and row-wise simultaneously, the correct choice stands out. We can see that C can be obtained from A by replacing the square with a circle. The only choice that can satisfy both A:B and A:C is Choice 2. This is because Choice 2 can be obtained by horizontally-replicating C and by deleting the squares in B and replacing them with circles, thus matching both the row and column transformation rules.

## 4. Code Design

The system has been built using the **C++** programming language. The code design closely follows the architectural block diagrams described in section 3.1. For image manipulation, I have made use of **OpenCV** and for blob

Here are the four main functions that have been used:

1. **cv::matchTemplate**: To perform the Normalized Cross-Correlation based Template Matching (used in sections 3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.1.5 and 3.1.6)

2. **cv::flip**: To flip the image, either in the horizontal or vertical direction (used in sections 3.1.4 and 3.1.5).

3. **cvFloodFill**: To fill the image with a specified color (used in section 3.1.6).

4. **cvBlobsLib**: To detect blobs/connected-components/sub-images within an image (used in section 3.1.3).

## 5. Experiments and Evaluations

During the course of the project, I ran several experiments to make sure that my system is behaving as expected. My initial experiments with Problem 1 (in Project 2 and now) made me realize that multiple choices could be correct and both row-wise and column-wise comparisons are necessary in order to pick the right answer. Thus, I had to introduce both row and column-wise comparisons to capture the multiple analogies. This added the first layer of complexity to the system because it now had to match multiple images against each other across rows and columns. Upon reflection, I realized that this is probably what, I as a human, am doing too. I am looking for all the possible analogies between A and B, both row-wise and column-wise and storing it in my memory and then matching each analogy with the possibilities for each choice.

My initial experiments led me to take a reflection and thought based approach. That is, whenever I had to make a design decision, I tried to work through my own thought processes and mimic them as best as I could.

Problem 2 presented the challenge of multiple shapes within a given evidence/choice. To handle it, I had to introduce blob-detection. It could have been probably done in a simpler way. But my motivation was to mimic human behavior as much as possible. We humans immediately perceive the figures to be made up of either one or more shapes. Thus, it made the most sense to use blob-detection in order to make the system aware that there are multiple shapes present. This gives it the capability to solve the problem like us.

Problem 3 presented the challenge of colored shapes. To handle this, I had to introduce a new attribute called fill-color and a new transformation module that checks for color changes.

The program runs fast. The average time from input to output is 0.72 seconds (720 milliseconds). Stage 1 takes up 99.9% of this time, where the bulk of the image manipulation and template matching occurs.

I would like to mention that the choice of the programming language and the right libraries helped me a lot. OpenCV with C++ provides functions that allows us to do fast image-manipulation and also allows one to build powerful constructs and complex data-structures that make building such systems easier. More time is spent on thinking and designing than on programming. However it is important to note that with such great power and flexibility comes the downside that the code can quickly become unreadable and unmanageable. But with proper care and adhering to good software engineering principles, this problem can be easily avoided.

## 7. Generalization

The system that I have designed can be easily generalized for future projects. Everything is designed so that it can be easily extended. Any new kind of transformations can be added to the list of transformation functions in the solve.cpp file.

Both the system architecture and the code have been designed to be modular. I have taken care to strictly follow the principle of single responsibility, i.e. each class and each function will do exactly the thing that it is designed to do. Because of such clean modularity, any new changes that have to be done can be made in the appropriate places without touching other parts of the code.

## 8. Conclusion

In this project, we have designed and built a system that is capable of solving 3 problems from the Miller's Intelligence Tests. The system architecture and code design was explained in detail. The system was thoroughly tested and was found to give the correct answers for all the six problems.