Assessing and Analyzing Tesseract Based Nepali Script OCR

Sudan Prajapati[1]*, Aman Maharjan[2], Shashidhar Ram Joshi[3], Bikash Balami[4]

[1]Department of Computer Science, Deerwalk Institute of Technology, Kathmandu, Nepal
sudan.prajapati@deerwalk.edu.np

[2]Central Department of Computer Science and Information Technology, Tribhuvan University, Kirtipur, Nepal
aman.maharjan@gmail.com

[3]IOE, Pulchowk Campus
srsrjoshi@ioe.edu.np

[4]Central Department of Computer Science and Information Technology, Tribhuvan University, Kirtipur, Nepal
bikuji@gmail.com

Abstract: Character recognition is commonly referred to as Optical Character recognition as it deals with the recognition of optically processed characters. With the advent of digital optical scanners, a lot of paper- based books, textbooks, magazines, articles, and documents are being transformed into an electronic version that can be manipulated by a computer. OCR is an instance of off-line character recognition, where the system recognizes the fixed static shape of the character. This paper focuses on character recognition of printed text in Nepali script. This work analyzes the efficiency of Nepalese OCR based on Tesseract engine. The benchmark of this investigation and analysis is to create the dataset of the 69 different fonts with the 2,484 samples of consonants data of Nepali script. The overall accuracy of 96% was obtained in the training phase and 69% in the testing phase.

Keywords: Optical Character Recognition, Nepali Script, Tesseract, Nepali Font, Character Recognition

1. Introduction

The recognition of the character by machines has been a research topic for decades. Before the age of digitized computers, not much research was done in the field of character

recognition. In early research works, printed character recognition generally used template matching; low-level image processing techniques were used on the binary image to extract feature vectors, which were then fed to statistical classifiers.

A mechanical or electronic translation of images of handwritten, typewritten or printed text into machine-editable text form is defined as Optical Character Recognition (OCR). Automatic text recognition using an OCR is the process of converting images containing text into the equivalent string of characters. OCR is one of the most challenging topics in the field of pattern recognition [1]. OCR technologies are used for various purposes like storing documents, searching text, information retrieval from paper-based documents, documenting library materials etc. OCR can be categorized into 3 types: offline handwritten text recognition, online handwritten text recognition and machine printed text recognition.

Table 1: Nepali Consonant Characters

क	ख	ग	घ	ङ	च	छ	ज	झ	ञ
ट	ठ	ड	ढ	ण	त	थ	द	ध	न
प	फ	ब	भ	म	य	र	ल	व	
श	ष	स	ह	क्ष	त्र	ज्ञ			

Table 2: Nepali Vowel Characters

अ	आ	इ	ई	उ	ऊ	ए	ऐ	ओ	औ	अं	अः

Table 3: Nepali Numeric Characters

०	१	२	३	४	५	६	७	८	९

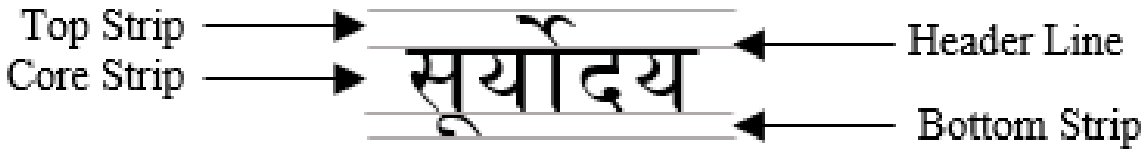


Fig.1: Nepali Word

Tesseract OCR engine is an open source OCR engine developed by the HP labs [2]. It was originally developed for English but later, extended to recognize other languages like Arabic, French, Italian, Catalan, Czech, Danish, Polish, Bulgarian, Russian Japanese, Chinese, Devanagari etc. Training the Tesseract OCR engine for any new script requires in-depth knowledge of the script and its character set. Tesseract needs a comprehensive and comparative study to deal with character recognition problem.

2. Problem Definition

Nepali is written from left to right direction. It is a phonetic and syllabic script, so, it does not have any lower or uppercase characters. The distinctive feature of Nepali script is the presence of a horizontal line on the top of all character known as the header line, shirorekha or diko [3]. The words can typically be divided into three strips: a core strip (middle zone), a top strip (upper zone), and a bottom strip (lower zone). When two or more characters appear side by side to form a word, the header lines touch and generate a bigger header line (figure 1).

Nepali character recognition using OCR faces many problems. One of the major problems is in segmenting characters. This problem arises due to the modifiers, combined characters, the variability of character size and so on [3].

Various approaches of character recognition can be applied to Nepali OCR such as gradient features, template-based formulation, and classification techniques like ANN, HMM, SVM are used [2-4].

3. Literature Review

The early attempt for Character Recognition was somewhat limited due to unavailability of powerful computing devices and digital image capturing devices [2]. Powerful hardware became commonplace during the 1980's which subsequently accelerated research in both online and offline OCR [1].

Image processing techniques and pattern recognition powered by artificial intelligence and various intelligent learning algorithms like ANN, HMM, fuzzy set reasoning etc. are commonly used in OCR [5]. The ultimate milestone of an OCR is to convert the printed or handwritten scanned document into machine-encoded text with negligible error. The Electronic device equipped with an OCR system can improve the speed of input operation and decrease possible human errors. OCR systems can decrease the use of keyboard and act as the interface between man and machine to a great extent. It also helps in office

automation leading to huge saving of time and human effort.

The Nepali language is written in the Devanagari script and hence the unique characteristics of the Devanagari script as reported in [5-7] apply to the Nepali printed text as well. Research in Devanagari OCR has been done extensively many people in the past. Similar achievements have been reported for the Bangla script which is similar to the Devanagari Script in many respects [6,7].

This study is focused on making use of the already available techniques in OCR with slight modifications necessary for the Nepali language.

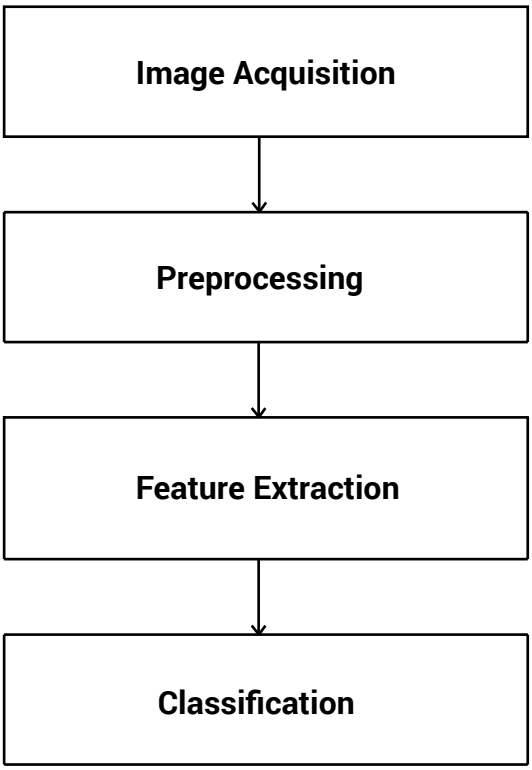


Fig.2: Steps of System Implementation

4. System Functioning

Hierarchical level of character recognition for the proposed system is divided into four subsections – image acquisition, preprocessing, feature extraction and classification (figure 2). The hierarchical model of is given below:

4.1. Image Acquisition

Images were acquired by typing characters in a word processor, printing and scanning in 300 DPI resolution. Character samples of different fonts were collected in PDF format.

4.2. Preprocessing

Raw image may contain some noisy pixels. They can be removed using the median filter. The resulting image is then ready for segmentation. Segmentation separates the digitized image into individual constituent character images.

4.3. Feature Extraction

After preprocessing of the image, feature vectors can be extracted for training and testing. Feature extraction plays an important role in the recognition of character to distinguish the character from image.

In Tesseract, the features for prototypes are 4 dimensional (x, y, angle, length) with 10-20 features in a prototype configuration. The features for unknown are 3 dimensional (x, y, angle) and for each character, there are approximately 50-100 features. The normalized features for the unknown are computed by tracing around the outline of the blob unit twice. The x, y position and the angle between the tangent line and a vector eastward from the center of the blob are saved as features.

4.4. Classification

Classification and recognition of a particular dataset depend on the selection of the features and classifiers that can recognize a particular character pattern. A classifier is called the 'heart' of pattern recognition system. It takes feature vectors and goes through all the decision-making process for recognition of patterns.

5. Training Tesseract

To train Tesseract for Nepali script, training data consisting of the following four files need to be created in the tessdata subdirectory of Tesseract installation directory:

- nep.inttemp
- nep.normproto
- nep.pffmtable
- nep.unicharset

The first three letters in each file, nep, represents the language code of Nepali in ISO 639-2 standard [8,9].

5.1. Training Data

Raw training data consists of 56 files, one for each Nepali font. Each file contains all the consonant characters in Nepali and was scanned from printed documents to 300 DPI TIFF

format.

5.2. Box File

Box file contains lists of characters, one per line, with the coordinates of the bounding box around the image. Initial box file generated by tesseract does not contain correct Unicode characters for Nepali, nor do they contain correct coordinates for bounding box (table 4). They are to be edited manually to fix this (table 4).

Table 4: Generated Box File

S	100	3147	159	3200	0
v	216	3146	273	3200	0
...	...	...	...	...	...

Table 5: Edited Box File

क	100	3147	159	3200	0
ख	216	3146	273	3200	0
...	...	...	...	...	...

5.3. Training File

Character images and corresponding box files are then used to create an exp.tr file, which contains the features of each character.

5.4. Character Set File Generation

Tesseract needs to know the following character properties: isalpha, isdigit, isupper, islower, ispunctuation before training the characters. This data is encoded in the unicharset file.

5.5. Font Properties

Font properties are used to provide font style information that will appear in the output when a font is recognized. This is provided in a text file names font_properties in the following format:

```
<font name> <italic> <bold> <fixed> <serif> <fraktur>
```

For a font named Mangal, the content of the file looks like:

```
Mangal 1 0 0 0 0
```

5.6. Clustering

After the extraction of character features of all the training pages, they are clustered using mftraining and cntraining commands. The mftraining command uses font_properties and unicharset files to generate inttempfile containing shape prototypes. The cntraining command generates the normproto data file that handles character normalization training for Tesseract.

5.7. Final Step

The final step combines the generated files in the previous steps. The required files are renamed by attaching a three-letter prefix, nep, for the Nepali language, to each file name.

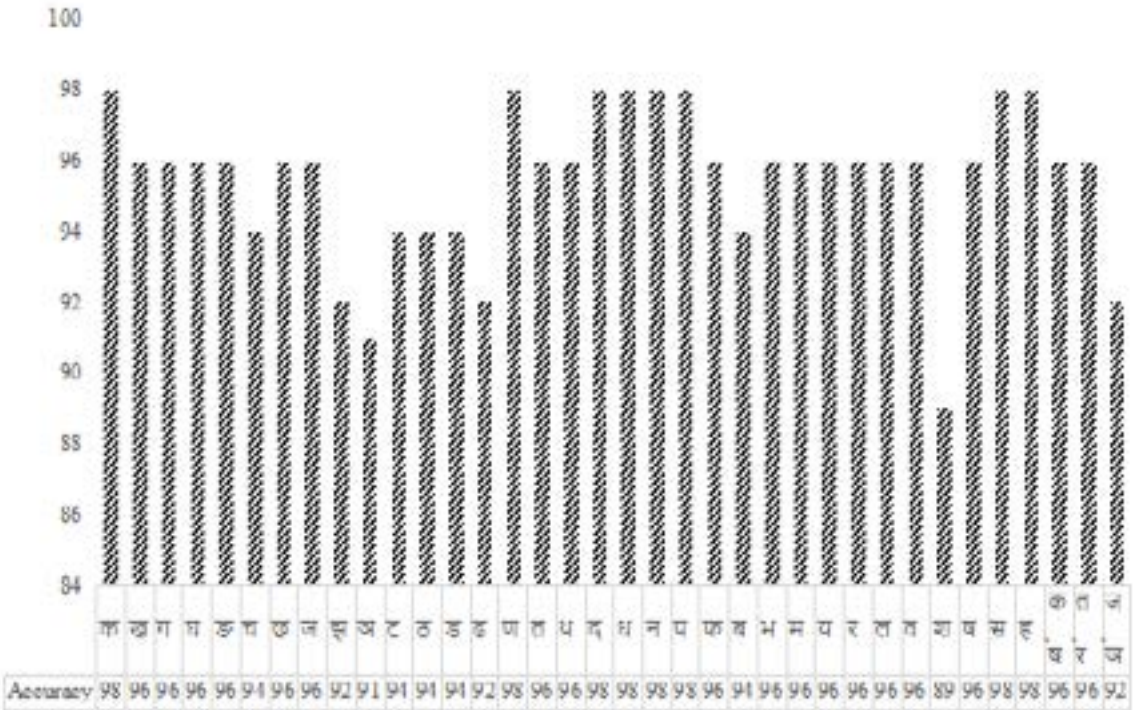


Fig.3: Training Accuracy for Individual Characters

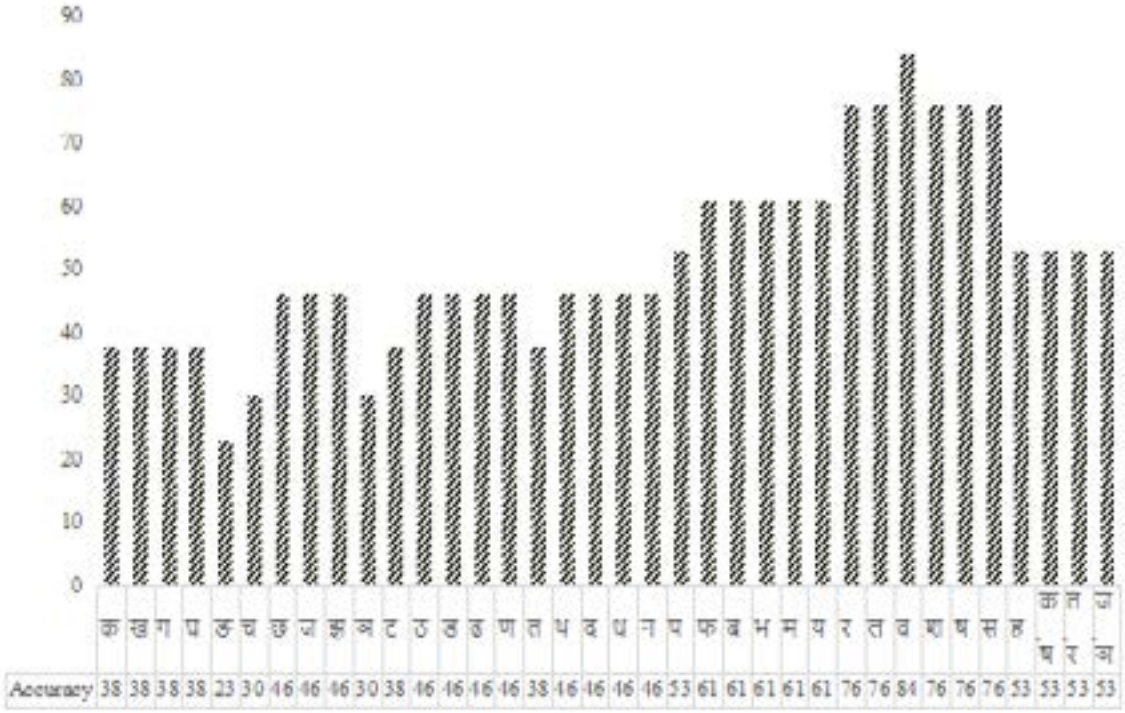


Fig.4: Testing Accuracy for Individual Characters

6. Evaluation and Result

The main aim of this study is to experimentally verify and evaluate the performance of Tesseract in optical character recognition of Nepali characters. The experiment is done in consonants of Nepali script, assuming the same procedure will be followed to the vowel and numerical digits. Dataset was created from fonts with 69 typefaces, containing 2520 individual character, for training and testing purposes. Training and testing were each carried out 10 times using random samples from the dataset.

Table 6 shows the training and testing performance for Tesseract. Each phase is carried out 10 times, and average values are shown in the table. 80% of the data was used to training and the remaining 20% was used for testing Tesseract engine. The result shows average training accuracy of 96% and average training time of 2.1 minutes. Similarly, it shows 69% average testing accuracy and 30 second average test completion time.

Similarly, individual character recognition accuracy for training data is displayed in figure 3 and for test data is displayed in figure 4. The figures show that overfitting is a major issue in Tesseract.

Table 6: Accuracy and Time

Phase	Average Time	Recognition Accuracy
Training	2.1 min	96%
Testing	30 sec	69%

7. Conclusion

Tesseract follows blob detection technique – it considers touching foreground pixels to be part of the same blob. For the individual character, a character component analysis is used to extract the character outline which is very useful because it does the OCR of an image with white text and black background. During the training phase, the segments of a polygonal approximation are used for features. In the recognition phase, features of a small, fixed length (in normalized units) are extracted from the outline and matched many-to-one against the clustered prototype features of the training data.

The Overall accuracy of 96% was obtained in the training phase and 69% in the testing phase. Similarly, average training time was 2.1 min and average test completion time was 30 sec. Analysis of individual character recognition data shows that Tesseract suffers from overfitting problem.

Character recognition, especially in a language like Nepali, has been a challenging research area for decades. Many research and various techniques have been carried out for the Devanagari script it uses, but due to the peculiarity of the script, the recognition accuracy has not been 100%. This can be addressed in future studies. Similarly, research on the word and sentence level Natural Language Processing (NLP) can also be carried out. A corpus of words can be created and used for translation. Handwritten Devanagari script recognition and multilingual character recognition is another comprehensive field of study for future researchers.

Reference

1. Y. Lu, Machine Printed Character Segmentation – an Overview, vol. 28, Pattern Recognition, 1995, pp. 67-80.
2. R. Smith, "An Overview of the Tesseract OCR Engine," Proc. of ICDAR 2007, 2007.
3. B. K. Bal, R. Pandey, S. Tuladhar and S. Shakya, "Interim Report on Nepali OCR," 2006.
4. M. Gunasekaram and S. Ganeshmoorthy, "OCR Recognition System Using Feed

Forward and Back Propagation Neural network," Department of MCA, Park College of Engg & Tech, Coimbatore.

5. D. Yadav, S. S. Cuadrado and J. Morato, "Optical Character Recognition for Hindi Language Using a Neural-Network Approcah," J Inf Process Syst, vol. 9, no. 1, 2013.
6. B. K. Bal, "Scripts, Segmentation and OCR II Nepali OCR and Bangala Collaboration," Jan 2009.
7. B. Chaulagain, B. B. Rai and S. K. Raya, "Final Report on Nepali Optical Character Recognition," 07 2009.
8. R. Smith, D. Antonova and D.S. Lee, "Adapting the Tesseract Open Source OCR Engine for Multilingual OCR," in Proceedings of the International Workshop on Multilingual OCR 2009, Barcelona, Spain, 2009.
9. "Codes for the Representation of Names of Languages," [Online]. Available: https://www.loc.gov/standards/iso639-2/php/code_list.php. [Accessed 2016-08-28].

Maze Solving with Anomaly Tolerant Dead End Filling and A*: Implementation and Analysis

Ruby Shrestha
Department of Computer Science, Deerwalk Institute of Technology, Kathmandu, Nepal
rshrestha@alumni.deerwalk.edu.np

Abstract: Maze solving, based on the concept of path finding and AI agent movement, has numerous solving techniques and approaches. This paper aims to propose, implement and empirically assess a unique mixed model approach to solve maze image uploaded by the user, and intends to show that the approach is feasible and effective. The mixed model implementation involves anomaly tolerant dead end filling using image processing followed A*. Along with the primary aim of empirically assessing mixed maze solving approach, this paper also aims to introduce distortion/anomaly tolerant version of dead end filling algorithm and show its usage in maze solving. The maze image uploaded by the user is first digitized and preprocessed, and is then solved using mixed model implementation. The overall system performance is tested using 10 different maze images of varied sizes, shape and number of dead ends. With a size of 733 x 700 pixels and 179 dead ends, a maximum total time of 34.151 seconds is obtained. Moreover, the correlations of image properties with proposed model components are also evaluated. It is observed that the approach works with various maze shapes provided they have only two openings, at the edge, and are infected by only salt and pepper noise, if any; hence, the proposed model for maze solving is generic and practical with some limitations.

Keywords: Path Finding, Automatic Maze Solving, Image Processing, Dead End Filling, Anomaly Tolerant, A*

1. Introduction

Agent movement is amongst the greatest challenges in the design of realistic Artificial Intelligence (AI) in computer games. Path finding strategies are generally utilized as the core

of such AI agent movement [1]. Path finding is a popular and frustrating problem in game industry [2]. Besides for the game industry, the domain of path finding comes with importance for various other working areas as well, such as logistics, operation management, system analysis and design, project management, network and production line [3]. Path finding is the way to avoid obstacles smartly and look for the most beneficial path over different places. Solving a maze can be a good example of avoiding obstacles smartly and looking for the most beneficial path over different places.

Mazes are network of paths designed as puzzles with complex branching passages through which a solver requires finding route [4]. Maze solving is simply finding a way out of such networks of paths designed as puzzles, starting from the entry point and stopping at the exit point. Maze solving is still considered as an important field of robotics [5]. Maze solving, a seemingly minor challenge for the analytical minds of humans, has generated enough curiosity and challenges for AI experts to make their machines solve any given maze [6].

For humans, maze is something that other humans have created, something that acts as a challenge to one of their most important skills: the ability to create an ecocentric cognitive map of their environment. Maze solving for them is hence navigating through that cognitive map [7]. Solving mazes for humans is a task requiring critical thinking. When provided with a maze, it requires them some time to figure a way out; this time increases significantly with increase in the complexity of maze. Automating the process of maze solving with computers can hence lead to significant minimization in the effort required by people in the process.

Moreover, automatic maze solving or navigation has a wide prospect and practical applications in real world: the top view of some manufacturing plant can be considered as a maze in which a robot might have to navigate; a blue print of a house or floor can be regarded as a maze in which some human might require finding some room; a network of water pipelines is synonymous to a maze in which one needs to decide the shortest path for water delivery or the placement of water stopper; road system is similar to a maze in which intelligent traffic control, determining the shortest path, can help vehicles to provide emergency services.

Realizing its prospect, different researches have been carried out in the domain of maze solving using computers. A variety of algorithms have been used to solve mazes and a variety of implementations have been compared. Researches, such as the one in [4], have compared Non-Graph Theory (NGT) algorithms, like Wall Follower, with Graph Theory

(GT) algorithms, like Flood Fill, Depth First Search (DFS), Breadth First Search (BFS), in the context of maze solving; these have concluded GT algorithms to be more efficient. There are different other NGT algorithms as well, such as dead end filling, image processing based approaches, and other GT algorithms, such as A*, IDA*, AO*, Uniform Cost Search that are involved in implementations, evaluations and comparisons. Moreover, mixed models (NGT and GT combined) are also available for maze solving.

However, research on mixed maze solving approach that combines dead end filling algorithm, which is **anomaly tolerant,** with GT based approach such as A* or similar is scant. So, the proposed approach for maze solving aims to contribute to that limited sphere of maze solving techniques. It uses mixed model to solve maze image uploaded by the user, be it prior produced or pressed maze; the mixed model involves Anomaly Tolerant Dead End Filling using Image Processing (IP), to find all possible paths, followed by Graph Theory based approach, A*, to find the shortest path. Besides the primary aim of developing and assessing the unique mixed model, this paper also aims to introduce anomaly tolerant version of dead end filling and highlight its use in the process of solving mazes. Anomaly Tolerant Dead End Filling algorithm is named so in the paper since it is the dead end filling algorithm which is updated in such a way that it is capable of handling irregularities or anomalies along the maze paths while solving it. Following an approach that is indifferent to anomalies available in maze paths can cause the implementation of algorithm to be ineffective in solving maze; it results in left-out paths and pixels or incorrect final path or such similar unexpected behavior or outcome.

In the proposed approach, the maze image uploaded by the user is first digitized by the system; it is then preprocessed using median filtering to remove salt and pepper noise, Niblack local thresholding to binarize the resultant image, and Zhang-Suen thinning to develop single pixel paths for further maze interpretation. The single pixel paths may not be ideal; some irregularities in position may occur after the preprocessing phase. Hence, following an anomaly tolerant approach for implementing dead end filling algorithm becomes necessary. The proposed approach thus uses anomaly tolerant dead end filling implementation to find all possible paths in the maze image, followed by A* to find the shortest path, if any, in the maze.

The paper explains the development of proposed model, evaluates the correlation of maze elements and features with the proposed model components, and shows that the proposed model is practical with some limitations. This paper is organized as follows: Section 2 discusses previous related works and the algorithms involved. Section 3 describes

the proposed design of the system and its implementation. Section 4 presents the empirical data obtained from 10 different runs of the proposed approach. Section 5 discusses and analyzes the empirical data and results. Finally, Section 6 summarizes the conclusions of this work.

2. Literature Review

2.1. Maze Solving Approaches

The great Swiss mathematician Leonhard Euler was one of the first to study mazes scientifically and in doing so founded the science of topology [8]. Unlike the days where a simple viewing of the maze was enough to solve them, mazes have now evolved along with their solving techniques [9]. Maze solving approaches are broadly classified as traveler-based and maze-based [10]. In traveler-based approach, the maze solver traverses the maze physically and hence, has no idea about the entire maze structure at once. In maze-based approach, one has idea about the entire maze structure at once [11]. At present, maze-solving with computers is gaining significant interest and therefore, maze-based approaches are found to be significantly used. The Random Mouse, Pledge, and Tremaux are some traveler-based approaches while GT algorithms like BFS, DFS, A* and NGT algorithms such as dead end filling and image processing based approaches fall under maze-based approaches.

2.2. Related Works - Automatic Maze Solving

2.2.1. Using Uninformed Search Techniques

Computer implementation of maze solving is widely dependent upon searching techniques. GT algorithms can be regarded as a subset of searching techniques. Topology and GT Mathematicians have been studying maze creation and maze solving for several centuries [8].

Most of the researches have compared and contrasted GT algorithms for maze solving with NGT algorithms. In [12], researchers have compared NGT algorithm (Wall Follower) with GT algorithms (DFS and Flood Fill) in the context of maze solving, and have concluded GT algorithms to be more efficient.

Moreover, in [9], comparison between different GT algorithms like flood fill, modified flood fill, BFS, and DFS have been carried out in the domain of maze solving. The research has concluded BFS to be a better alternative with respect to average time consumption,

dead end handling, completeness, and optimality for large mazes.

Although BFS has been concluded to be a better alternative among other GT algorithms for maze solving, it is an uninformed search technique; informed search techniques are considered to be almost always more efficient than uninformed searches, and often more optimal [13].

2.2.2. Using Informed Search Techniques

There are a number of informed search techniques. Hill Climbing, A*, AO*, and other variants of A* such as IDA*, Dynamic A*, Lifelong Planning A* are some examples of informed search techniques among many others.

In [14], hill climbing search has been analyzed for the purpose of solving Sokoban problem. Sokoban problem is a Japan-based transport puzzle which is very similar to maze problem or it can be regarded as a Japanese version of the maze problem. [14] mentions that hill climbing search is not applicable for solving Sokoban problems, except very easy ones. This implies that its use for maze solving is also not much reasonable.

Furthermore, according to Yoshitaka Murata and Yoshihiro Mitani in [15], A* outperforms Dijkstra in the context of maze-solving when keeping search time into consideration. This indicates that A* is a more efficient alternative to Dijkstra for the purpose of solving mazes.

As noted in [16], AO* is an AND-OR search technique. It is applicable to AND-OR graphs or trees. However, maze is an OR graph problem and hence, cannot be solved using AO* search technique. Moreover, with IDA*, due to its inherent properties of calculating arc cost and heuristic function, the nodes closer to the source square are revisited many times hence slowing down its performance in context of maze solving [17]. This indicates that IDA* is also not much preferable. Other variants of A* do not have a rigid, adequate evidence of their usability or reliability in context of maze solving.

Although multiple researches yield an implication that A* is one of the most efficient search techniques for maze solving, [18] demonstrates that maze solving using IP technique along with path finding algorithm (GT) is faster because it acquires maze's data beforehand rather than simply traversing through each available path in the course of solving it. To add, Behnam Rahnama, Atilla Elçi & Shadi Metani [19], based on their research, concluded that IP algorithms, when used in mazes, give agent preplanning time and helps to avoid mistakes such as loop or dead ends.

2.2.3. Image Processing

There are various Image Processing approaches for solving mazes. Dead end filling approach is considered to be always very fast, and uses no extra memory [20]. This includes filling in passages that become parts of dead ends once other dead ends are removed. Before processes such as dead end filling can be carried out on any image, they need to be preprocessed or enhanced.

Image enhancement in IP consists of variety of techniques to improve the visual appearance of an image, or to convert the image to a form that is better suited for human or machine interpretation [21]. De-noising, Thresholding, and Thinning are some common preprocessing / enhancement techniques for maze images.

- De-noising

  Salt and pepper noise is one of the most common types of random noise that is likely to be encountered in images [22]. Median filtering is a nonlinear process useful in reducing such impulsive or salt-and-pepper noise [23].

- Thresholding

  According to [24], local threshold method performs better than global threshold method in case of badly illuminated images and document image analysis as threshold computation is dependent on region characteristics. [24] compares various local threshold methods such as Niblack, Sauvola, Bersen, Feng's techniques of local thresholding using various images and concludes Niblack local thresholding to be better.

- Thinning

  The application of thinning to maze images helps to solve them quickly [15]. Thinning the image uploaded by the user actually reduces the search space. As noted in [25], there are many thinning algorithms available, and all have their own advantages and disadvantages. According to [25], the choice of the thinning algorithm depends on the application, as not all thinning algorithms will be suitable for particular application. However, it is to note that Zhang-Suen, one of the parallel thinning algorithms, has faster processing time in comparison to Guo and Hall's algorithm.

3. Algorithm Development

The algorithm consists of several steps, starting with image capture, followed by image preprocessing, points of interest determination, anomaly tolerant dead end filling, A*, and finally ending with solution overlay. The algorithm is developed following a general

approach so that it works with mazes of different shapes provided that they are two dimensional or top-view of three dimensional mazes. The algorithm requires start, end, boundaries and paths of the maze to be clearly visible. As salt and pepper is one of the most common types of random noise that is likely to be encountered in images [22], the algorithm has been designed to handle salt and pepper noise; other varieties of noise can result in wrong output. Moreover, the algorithm has been scoped out to support only those maze images with start and end at the edge. The block diagram of the overall algorithm is shown in Fig. 1.

Fig.1: Overview of the proposed algorithm for maze solving

3.1. Capture Image

The maze image uploaded by the user via the user interface provided in the system is first captured and digitized.

3.2. Preprocess Image

The maze image so captured and digitized is then preprocessed undergoing three major preprocessing steps: noise removal / filtering, binarization / thresholding, and path thinning.

3.2.1. Median Filtering

Median Filtering is used to remove salt and pepper noise. Median filtering replaces the center pixel of the window considered by median of the window. The window requires being some odd x odd value. It was realized through testing that higher the window size more is the blurring effect. So, the proposed algorithm uses 3x3 window for comparatively less blurring and better result. It is to note that in order to ease the implementation of median filtering, the colored image was changed into gray scale so that for each pixel, only gray

level values can be dealt with instead of red, green and blue values. To determine the gray scale image, the formula in (1), which was suggested by CCIR (Center for Clinical Imaging Research) 601[26], was used since it works well for the project:

Gray = 0.299 R + 0.587 G + 0.114.....(i)

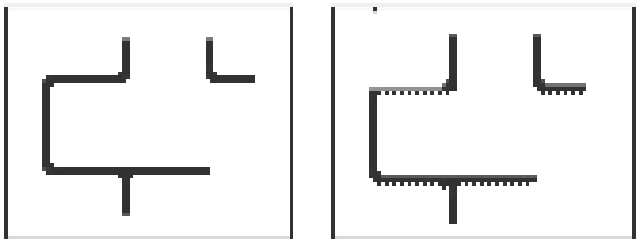
3.2.2 Niblack Thresholding

Niblack thresholding is a chosen method for local adaptive thresholding. It adapts the threshold according to the local mean and the local standard deviation over a specific window size around each pixel location [24]. The local threshold at any pixel (i, j) is calculated as:

$T(i, j) = m(i, j) + k\,\sigma(i, j)$.....(ii)

, where, $m(i, j)$ and $\sigma(i, j)$ are the local sample mean and standard deviation, respectively. The size of the window is dependent upon application. A window of size 3x3 has been chosen for better locality and effectiveness. The value of the weight 'k' is used to control and adjust the effect of standard deviation due to objects features. According to [27], if the value of k is increased keeping the window size same, some thinning effect is seen in the image during binarization.

Two different values of k (0.1 and 0.5) were tried out on a maze image. In case of k=0.5, some thinning attempt was noticed in the form of some irregularities in the boundary as shown in Fig. 2. Hence, k=0.1 was chosen.



(a) (b)

Fig.2: Difference in outcomes of two different values of k (a) k=0.1 and (b) k=0.5

3.2.3. Zhang Suen Thinning

The step of thinning was introduced in the proposed algorithm's preprocessing phase since it reduces the path search space and hence, eases the dead end filling process and in overall, eases finding solution to the search space. The thinning process should result in single pixel connected paths. Zhang-Suen Thinning is one of the popular parallel thinning algorithms and it guarantees connectivity and one-pixel wide solution [28].

3.3. Determine Points of Interest

In order to fill the passages that are part of dead ends, Points of Interest (PoIs) need to be initially determined. PoIs include junctions, dead ends, start and end pixels. Junctions, dead ends, start and end pixels are each distinct based on their number of neighbors. Junctions have greater than two path-colored neighbors while dead ends have one; start and end pixels are basically dead ends at the edge. The algorithm to find PoIs can be explained by Algorithm 1:

3.3.1. Algorithm 1

Until all pixels are traversed, for each pixel:

1. Let initially, pc-neighbors= 0,
   PX be the list of pixel values of 8 surrounding neighbors,
   0=black pixel, assuming path to be white colored and background to be black
2. For each N, where N=1…8
   If (PX[N] !=0 and PX[N+1]==0): pc-neighbors ++
3. If pc-neighbors >2: category = junction
   Else if pc-neighbors ==1: category = dead end
   Else: category = path pixel

Note: Time Complexity of Algorithm 1= O(P), where P is the number of pixels in maze image.

3.4. Fill Dead Ends: Anomaly Tolerant Dead End Filling

Standard dead end filling algorithm can be explained by Algorithm 2. It follows an iterative approach until all the dead end paths are removed from the maze. Each dead end pixel (except start and end) is traversed and in each traversal, single path associated with the dead end is removed, that is, a path from dead end to junction. Removing some dead ends can lead to other dead ends hence an iterative approach is followed to get rid of all the dead ends in the maze.

3.4.1. Algorithm 2

Until all dead ends are removed:

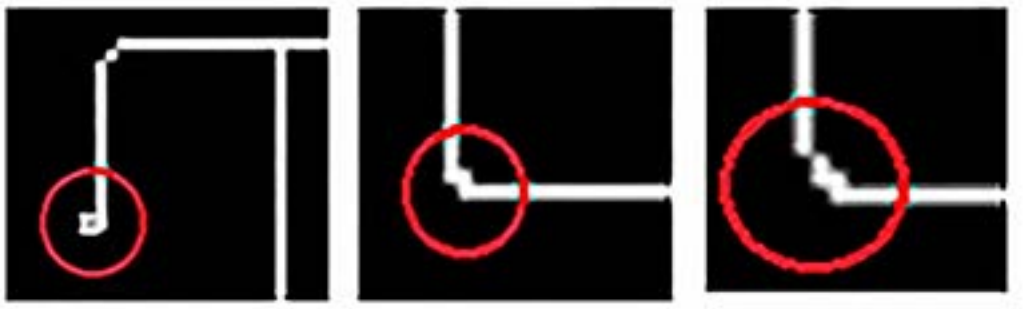
1.D=dead ends in Current_Maze – start and end pixels

2.New_Maze=maze obtained by removing single path associated with each dead end in D

3.Current_Maze =New_Maze

Special attention needs to be placed on point 2 of Algorithm 2. It needs to be implemented in such a way that it is able to handle and fill some irregularities that can appear after thinning algorithm is performed on the image.

Once thinning algorithm is performed on the maze image, some distortions are possible to occur. Such distortions can also be termed as anomalies since they result in dead ends, junctions and path pixels to have irregular number of path-colored neighbors.



(a) (b) (c)

Fig.3: Different anomalies that can occur in thinned maze's PoIs as shown in (a), (b) and (c)

Fig. 3 shows various anomalies that can appear in the maze image after thinning is performed. Here, 8-neighborhood is considered as local region. Fig. 3, (a) is distortion affected dead end with two path-colored neighbors, (b) is distortion affected path pixel with four, and (c) is also distortion affected path pixel with three path-colored neighbors. If dead end filling algorithm is implemented overlooking such irregularities, the implementation can result in unexpected outcomes such as left-out paths or pixels and incorrect final paths. Therefore, in order to handle such distortions, point 2 of Algorithm 2 can be replaced by Algorithm 3:

3.4.2. Algorithm 3

Considering path pixels as being white, for each dead end:

1.P=dead end pixel

2.N=P's neighbors with same color as it is

3.Change P to black / background color

4.If size(N)>4: stop with error as size(N) cannot be >4 after thinning is performed

5.Else :

a. Remove previous pixels from N

b. If size(N) after removing previous pixels is >1, choose as successor, the pixel with either same x coordinate as P or same y coordinate as P; else, successor is the remaining N after step (a)

c.Add P to list of previous pixels

d.P=successor

6.If first junction reached, start from step (1) with the next dead end in dead end list D; else, go to step (2)

Note: Time Complexity of Algorithm 3= O(NM), where N= No. of dead ends, M=Number of pixels in each dead end path.

Algorithm 3 is the anomaly tolerant version of dead end filling algorithm. Algorithm 2 and 3, together, results in maze with all possible paths.

3.5. Find Shortest Path: A* Search

A* is one of the popular algorithms by N.J.Nilsson, B. Raphael and P.E.Hart [29] for finding the shortest path from among a number of possible paths. A* has a significant contribution in the domain of path finding in AI. The proposed algorithm prefers A* to find shortest path over other algorithms as per the information suggested in literature review (Section 2).

4. Result and Observation

The proposed algorithm is implemented using Python and its libraries such as OS, Numpy, OpenCV. Ten different maze images are run through the system developed in order to test whether the system works correctly for each one of them. Mazes of three different shapes: rectangular, circular and hexagonal were tested. The system worked as expected for each one of them.

The time taken for preprocessing and post-processing were also noted. Each time value was calculated as average of three consecutive time values obtained when executing the program. Table 1 indicates how time taken varies with variation in the image size (number of pixels) and number of dead ends.

Table 1: Time Taken by Different maze images

SN	Image Size (pixels)	Number of Dead Ends	Preprocessing Time (sec)	Dead End Filling Time (sec)	A* Time (sec)	Total Time (sec)
1	170 x 189	11	1.106	1.102	-	2.208
2	223 x 226	16	1.502	1.860	1.902	5.264
3	287 x 319	20	3.438	2.914	-	6.352
4	284 x 277	24	3.486	3.195	-	6.681
5	354 x 360	77	5.726	6.030	2.341	14.097
6	582 x 302	39	6.917	5.304	5.311	17.532
7	596 x 663	96	11.251	14.631	-	25.882
8	543 x 570	42	9.017	7.848	14.049	30.934
9	580 x 577	74	13.735	13.745	5.366	32.846
10	733 x 700	179	16.380	17.771	-	34.151

Note: '-' is shown in A* Time column for those maze images which do not contain multiple paths. Table 1 is arranged in ascending order with respect to Total Time in seconds.

5. Analysis and Discussion

From Table 1, it can be comprehended that, in general, as the dead ends increase, the dead ends filling time also increase. However, consider image (5) and image (9). Here, although (5) has greater number of dead ends than (9), (9) has greater time. It is because the dead end paths in (5) are shorter, that is, contain less number of pixels than in (9). Such nature of dead end paths also affect the time required.

Based on Table 1, considering fields No. of Dead Ends and Dead End Filling Time (in sec), the graph in Fig. 4 can be constructed.

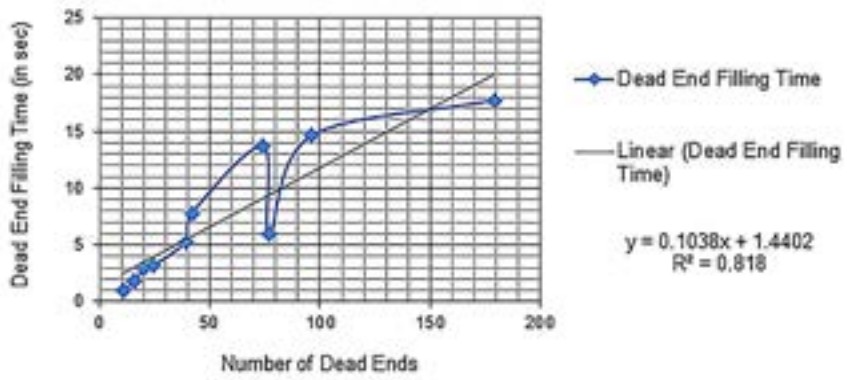


Fig.4: Graph of Dead End Filling Time (in sec) versus Number of Dead Ends

Approximately linear line can be constructed between number of dead ends and dead ends filling time with R^2 of 0.818.

Moreover, image size (in pixel square) is seen to be approximately directly proportional to preprocessing time. Based on Table 1, considering fields SN and Preprocessing Time (in sec), the graph in Fig. 5 can be constructed. In Fig. 5, Image Number is used to represent Image Size.

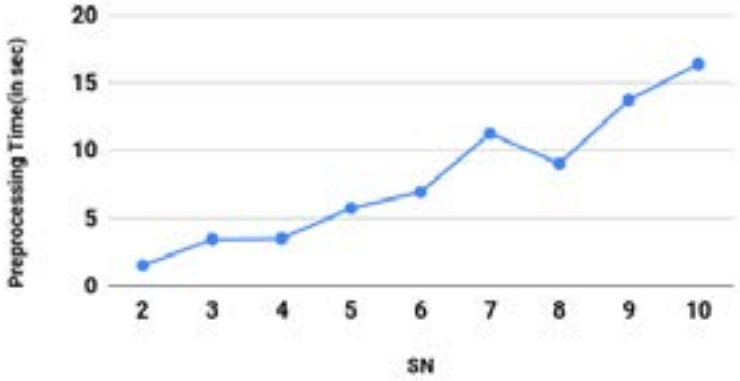


Fig.5: Graph of Preprocessing Time (in sec) versus Image Number (SN)

When A* time is considered, it is understood that A* time primarily depends upon the heuristic chosen since it is heuristic that directs which path to proceed next in each iteration. Next, it also depends on the number of pixels (graph nodes, in terminology of GT) to be traversed from source to destination.

On a different note, the number of possible paths is not a determining factor for A* time as observed from Table 2.

Table 2: Comparing number of possible paths with A* time

SN	No. of Possible Paths	A* Time (in sec)
2	4	1.902
5	2	2.341
6	2	14.049
8	4	5.366
9	2	5.311

These same concepts can be better understood with the correlation values in Table 3, which shows the correlation between different related categories.

Table 3: Correlation between different related categories

Categories	Correlation
Number of Dead Ends with Dead End Filling Time	0.90442
Image Size (in pixel square) with Preprocessing Time	0.97033

Table 4: Cross-correlation table

Categories	No. of Dead Ends	Image Size	Total Time
No. of Dead Ends	1		
Image Size	0.87342	1	
Total TIme	0.74749	0.94154	1

Table 4 is constructed in order to determine multiple correlation coefficient between the set {number of dead ends, image size} and total solving time. The multiple correlation coefficient, based on the standard formula, was determined to be 0.954.

6. Conclusion and Recommendation

The mixed maze solving model is successfully developed, implemented and tested using Flask framework of Python, in 4GB RAM based computer. Testing the implementation on ten different maze images shows that the model is feasible and effective for solving mazes within the scope of the project, that is those mazes with two openings at the edge and

infected by only salt and pepper noise. The maximum total time obtained was 34.151 seconds for 733 x 700 pixel maze image with 179 dead ends. Since the aim of this paper is not to indicate that the approach so described is better than other approaches that have been originated in the past, the paper has been limited to proposing, implementing and empirically assessing mixed model approach and showing that such an approach which involves a distortion tolerant version of dead end filling, combined with A*, is also feasible, effective, and hence usable approach for solving mazes.

Moreover, when the mixed model is empirically assessed, the results reveal relations between various maze entities. The number of dead ends and dead end filling time are approximately linearly related with R square of 0.818. Besides the number of dead ends, length of dead end paths also affects dead end filling time with the proposed implementation. To add, image size is approximately directly proportional to preprocessing time. A* time, which depends on heuristic chosen, does not depend upon number of possible paths. The set {number of dead ends, image size} and total maze solving time is highly correlated with a multiple correlation coefficient of 0.954.

Possible future works involve optimizing the implementation of the proposed model. The implementation can be optimized using new software technologies such as machine learning, genetic algorithm or functional programming. In addition, the scope can be enhanced to handle noises besides salt and pepper, and multiple starts and ends.

Reference

1. R. Graham, H. McCabe and S. Sheridan, “Pathfinding in Computer Games”, The ITB Journal, vol. 4, no. 2, pp. 57-81, 2003.
2. X. Cui and H. Shi, “A*-based Pathfinding in Modern Computer Games”, IJCSNS International Journal of Computer Science and Network Security, vol. 11, no. 1, pp. 125-130, 2011.
3. N. Barnouti, S. Al-Dabbagh and M. Sahib Naser, “Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm”, Journal of Computer and Communications, vol. 04, no. 11, pp. 15-25, 2016.
4. J. Pandian, R. Karthik and B. Karthikeyan, “Maze Solving Robot Using Freeduino and LSRB Algorithm”, International Journal of Modern Engineering Research (IJMER), pp. 92-100, 2017.
5. M. Alsubaie. “Algorithms for Maze Solving Robot.” Bachelor thesis, Manchester

Metropolitan University, Manchester, 2017

6. M. Chand, M. Goel and S. Rathore, "Maze Solving Algorithms." Internet: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.4944&rep=rep1&type=pdf, Mar. 20, 2016 [Aug. 18, 2017].
7. E. Weissmann , "Amazing Maze: What Science Says About Solving Labyrinths." Internet: https://news.nationalgeographic.com/news/2014/07/140730-science-mazes-labyrinth-brain-neuroscience/, Jul. 31, 2014 [Aug. 18, 2017].
8. T. De, "The Inception of Chedda: A detailed design and analysis of Micromouse.", University of Nevada, Las Vegas, Las Vegas, 2004.
9. M. Tak and S. Datta, "A Comprehensive and Comparative Study of Maze-Solving Techniques by implementing Graph Theory- implementation of Djikstra's algorithm for solving a maze", International Journal of Engineering Trends and Technology, vol. 28, no. 2, pp. 61-64, 2015.
10. N. Yew, K. Tiong and S. Yong, "Recursive Path-finding in a Dynamic Maze with Modified Tremaux's Algorithm", International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering, vol. 5, no. 12, pp. 1-1, 2011.
11. B. Gupta and S. Sehgal, "Survey on techniques used in Autonomous Maze Solving Robot", 2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence), 2014.
12. K. Sharma and C. Munshi, "A Comprehensive and Comparative Study Of Maze-Solving Techniques by Implementing Graph Theory", IOSR Journal of Computer Engineering (IOSR-JCE), vol. 17, no. 1, pp. 24-29, 2015.
13. P. Norvig and S. Russell, Artificial Intelligence: A Modern Approach, 2nd ed. Upper Saddle River, New Jersey: Prentice Hall, 2003, pp. 94-136.
14. P. Jarušek and R. Pelánek, "Human Problem Solving: Sokoban Case Study", FI MU Report Series, pp. 4-4, 2010.
15. M. Yoshitaka and M. Yoshihiro. A Study of Shortest Path Algorithms in Maze Images. In SICE Annual Conference, 2011.
16. A. Abahai T., "Optimized AO* Algorithm for and-OR Graph Search", IOSR Journal of Computer Engineering (IOSR-JCE), vol. 17, no. 4, pp. 124-127, 2015.
17. R. Marín, A. Bugarín, E. Onaindía and J. Santos, Current Topics in Artificial Intelligence, 1st ed. Berlin Heidelberg: Springer-Verlag GmbH., 2006.
18. O. Kathe, V. Turkar, A. Jagtap and G. Gidaye, "Maze solving robot using image

processing", 2015 IEEE Bombay Section Symposium (IBSS), 2015.
19. B. Rahnama, "An image processing approach to solve Labyrinth Discovery robotics problem", in 36th International Conference on Computer Software and Applications Workshops, Izmir, Turkey, 2012.
20. S. Sonavane and N. Choubey, "Maze solver with Imaging", Shirpur, India, 2014.
21. B. Chitradevi and P. Srimathi, "An Overview on Image Processing Techniques", International Journal of Innovative Research in Computer and Communication Engineering, vol. 2, no. 11, pp. 6466-6472, 2014.
22. R. Adhami, P. Meenen and D. Denis Hite, Fundamental Concepts in Electrical and Computer Engineering with Practical Design Problems, 2nd ed. Boca Raton, Florida: Universal Publishers, 2017, p. 497.
23. M. Nagu and N. Shanker, "Image De-Noising By Using Median Filter and Weiner Filter", International Journal of Innovative Research in Computer and Communication Engineering, vol. 2, no. 9, pp. 5641-5649, 2014.
24. M. Chandrakala, "Quantitative Analysis of Local Adaptive Thresholding Techniques", International Journal of Innovative Research in Computer and Communication Engineering, vol. 4, no. 5, pp. 8432-8439, 2016.
25. N. Khanyile, J. Tapamo and E. Dube, "A Comparative Study of Fingerprint Thinning algorithms", in Information Security South Africa Conference, Johannesburg, South Africa, 2011
26. D. Salmon, The Computer Graphics Manual. Springer-Verlag London Limited, 2011, pp. 1001-1004.
27. J. He, Q. Do, A. Downton and J. Kim, "A comparison of binarization methods for historical archive documents", Eighth International Conference on Document Analysis and Recognition (ICDAR'05), 2005
28. T. Y. Zhang and C. Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns", Communications of the ACM, Vol. 27, No. 3, 198
29. Q. Acton, Algorithms-Advances in Research and Application. Atlanta, Georgia: ScholarlyEditions, 2013, p. 412.