Solving Sudoku Through Android Vision
“Where Did I Go Wrong?”

Gareth Miller
B.A.I. Engineering
Final Year Project April 2013
Supervisor: Fergal Shevlin

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

____________________________________________________________________  __________
Name                                                      Date
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

______________________________  _________________________
Name                        Date
I would like to thank my supervisor Fergal Shevlin for all of his assistance during the undertaking of this project. His advice and guidance proved invaluable with regards to the completion this project.

I also wish to thank my parents for the support they offered me throughout this project and for their assistance in proof reading this report.
Abstract

The purpose of this project was to design an application for the Android mobile operating system that allows the user to take a photo of a partially completed, printed, Sudoku puzzle, e.g. from a newspaper, and have the application solve the puzzle for them while highlighting if the user made a mistake in their attempt to solve the puzzle. The finished application successfully extracts the relevant information from a photo and solves the original printed Sudoku. It also offers a number of options to the user to deal with their handwritten numbers on the puzzle when the application failed to correctly identify the characters due to differing handwriting styles.

This report outlines the process involved in creating the application, from the initial stages, right through to publishing the finished product on the Google Play Store for the world to use.
# Table of Contents

Chapter 1: Introduction

1.1 What is a Sudoku? ................................................................. 1
1.2 Motivation ............................................................................. 2
1.3 State of the Art ...................................................................... 2
1.4 Objectives ............................................................................ 3
1.5 Overview of the Android OS .................................................... 4
  1.5.1 Android Versions ............................................................ 4
1.6 Libraries Used ...................................................................... 5
  1.6.1 OpenCV ...................................................................... 5
  1.6.2 Tesseract .................................................................... 5

Chapter 2: Background

2.1 Edge Detection ..................................................................... 6
  2.1.1 Canny Edge Detector .................................................. 7
2.2 Contours .............................................................................. 9
2.3 Hough Lines ....................................................................... 10
2.4 Perspective Transformation .................................................... 12
2.5 Optical Character Recognition ................................................ 14

Chapter 3: Implementation

3.1 Step 1: Capture Photo .......................................................... 16
3.2 Step 2: Sudoku Extraction ..................................................... 18
  3.2.1 Edge Detection .............................................................. 18
  3.2.2 Contour Detection – Find Border of Sudoku .................. 18
  3.2.3 Correcting Sudoku Distortion ....................................... 20
3.3 Step 3: Extracting Printed Numbers ........................................ 21
3.3.1 Manually Correcting Recognition Errors ........................................... 24
3.4 Step 4: Solve Sudoku ........................................................................... 24
3.5 Step 5: Dealing with Handwriting ......................................................... 26
3.6 Step 6: Display Numbers .................................................................... 27
Chapter 4: User Interface (UI) .................................................................. 28
4.1 Imageview ............................................................................................ 29
4.2 Buttons and handlers ............................................................................ 29
4.3 Invisible Buttons used to select individual cells .................................... 29
4.4 Progress Bar ......................................................................................... 32
4.5 Toasts .................................................................................................. 33
4.6 Status Indicator .................................................................................... 33
Chapter 5: Testing ...................................................................................... 35
Chapter 6: Bringing to Market .................................................................... 37
6.1 Issues .................................................................................................. 38
Chapter 7: Conclusion ................................................................................ 40
7.1 Further Development ............................................................................ 40
Chapter 8: Bibliography .............................................................................. 42
Chapter 9: Appendix .................................................................................. 44
9.1 Contents of attached CD ....................................................................... 44
9.2 Link to the finished application on the Google Play Store .................... 44
9.3 “Where Did I Go Wrong?” Changelog ................................................ 44

VI
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Example of a Sudoku</td>
<td>1</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Overview of Canny Edge Detection Algorithm [8]</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Normal Image (left), Edge Detected (centre), Contours Highlighted (right)</td>
<td>9</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Picture of a square and its resulting Hough Space (Colours inverted for clarity)</td>
<td>11</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Local maxima highlighted in Hough Space with their corresponding lines [11]</td>
<td>11</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Distorted Image with Perspective Transform Performed</td>
<td>13</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Implementation Flowchart</td>
<td>16</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Application Code Structure</td>
<td>17</td>
</tr>
<tr>
<td>Figure 9</td>
<td>RGB image and resulting edge image produced using Canny() function</td>
<td>18</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Houghlines() on entire edge image</td>
<td>19</td>
</tr>
<tr>
<td>Figure 11</td>
<td>All contours highlighted (left) and largest contour highlighted (right)</td>
<td>20</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Largest contour extracted</td>
<td>20</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Angled Sudoku extracted (left), Outer lines highlighted (centre), Perspective warped (right)</td>
<td>21</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Image segmented into cells (left), Each cell recognised with no constraints (right)</td>
<td>22</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Only recognising numbers (left), Only recognising cells with a number in them (right)</td>
<td>23</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Individual Cell (left), Largest contour in cell highlighted by rectangle (centre), Result when largest contours recognised (right)</td>
<td>24</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Green indicates high quality recognition, yellow low quality. Red boxes show recognition results not matching solution (original number recognised in red, correct solution in yellow)</td>
<td>26</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Sudoku with Hints Displayed (left) Completed Solution (right)</td>
<td>27</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Overview of UI</td>
<td>28</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Buttons for both activities</td>
<td>29</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Calculations required to get scaled image size</td>
<td>30</td>
</tr>
</tbody>
</table>
Figure 22 - Buttons scaling correctly to Sudoku cells (shown as visible for demonstration purposes)........................................................................................................................................ 31
Figure 23 - Progress bar showing background activity being performed .......................... 32
Figure 24 - Toasts indicating result of solving algorithm .................................................. 33
Figure 25 - Status Indicators ............................................................................................. 33
Figure 26 - UI of ImageManipulations activity shown on a device .................................. 34
Figure 27 - Results of number recognition method tests .................................................. 36
Figure 28 -
Figure 29 - Application from start to finish ........................................................................ 41
Chapter 1: Introduction

1.1 What is a Sudoku?

A Sudoku is a very popular printed puzzle consisting of 81 individual squares, or cells, arranged in a 9 x 9 format. The 9 x 9 main cell block is subdivided into nine 3 x 3 blocks of cells (boxes). The rules for what these cells contain are defined as follows:

- Each cell must contain a number between one and nine.
- Each column and row of 9 cells can only contain one instance of any one number.
- Each of the boxes can only contain one instance of each number.

![Example of a Sudoku](image)

The cells of the Sudoku are initially mostly empty with only a few of them filled in as clues to the person trying to solve the puzzle. The object of the puzzle is to fill in the remaining empty cells so that the grid conforms to the rules outlined above.

The process of solving the Sudoku is purely logical. The initially filled cells should provide the solver with enough information to insert at least one other number without having to resort to guess work.
1.2 Motivation

If the user goes wrong somewhere when solving a Sudoku it can be very hard to recover and figure out where the mistake was made. More often they can get stuck on a puzzle and can’t see where to put the next number. Because Sudokus are logical puzzles this makes them easily solvable by computers. There are many applications available online that allow a user to input each number from their Sudoku which will then solve the puzzle for them. However this is time consuming. The motivation for this project is to make this process as easy as possible for the user.

Smartphones have become increasingly powerful in recent years and are easily capable of the simple calculations required to solve any Sudoku through a variety of methods. Because of this, it was decided that a system would be designed which can extract the relevant information from an image of a partially completed Sudoku and then pass this information to an algorithm that will solve the puzzle. This would mean that the person seeking the solution to the puzzle would simply need to use the camera on their phone to take a picture of the newspaper and allow the application to take care of everything else.

1.3 State of the Art

There are many free and paid Sudoku solving applications on the internet or available as applications for Android smartphones. The biggest drawback is that the vast majority of them require one to manually input the numbers from the Sudoku into them. This is not very user friendly.

The most popular application on the Google Play Store that solves Sudokus is called Sudoku Solver [1]. This application requires the user to input the numbers into the application each time they want to solve a Sudoku. The application works as expected solving any Sudoku that was inputted into it. It has over 10,000 active installs from the Google Play Store showing that there is a market for this type of application.

One application that utilises the phone’s camera to recognise the Sudoku is AR (Augmented Reality) Sudoku [2]. This free application is capable of detecting a Sudoku grid in a live video stream from the camera and then augments the solution to the puzzle in the video feed. This application was tested before starting this project and a number of issues arose. Firstly, because the solution is augmented on the video stream, the
numbers tend to jump around due to user’s hands shaking slightly while videoing the page. To counter this, the user may use two hands on the phone to hold it steadier resulting in them not being able to write down the solution. Processing a Sudoku as a video stream is not necessary when one image is all that is needed to solve the puzzle. The application also states that every number from 1 to 9 need to appear at least once in order for its algorithm to solve the puzzle. This is not an acceptable constraint because a Sudoku doesn’t need to have every number displayed once in order for it to be a solvable Sudoku. Another problem that was noticed was with consistent detection; this application did not work very well in low lighting conditions and sometimes it failed to recognise a Sudoku at all.

1.4 Objectives

This project is aimed at developing a user friendly application to assist the solving of Sudokus. The goal was to use computer vision to extract the relevant information from an image of a Sudoku that will then be passed to a Sudoku solving algorithm. Once the Sudoku is solved, the user will be able to decide whether they want to view the full solution, or just a hint of the next number to insert.

One novel factor for this application is that is aims to accommodate Sudokus that have handwritten input in them. The application attempts to recognise and differentiate between the printed and handwritten digits on a Sudoku. However it also offers the user the option to change the handwritten digits if they are recognised incorrectly or to ignore them completely when running the solving algorithm. This allows the application to highlight the cells in which the solved number did not match the handwritten number thus showing the user where they have made a mistake.
1.5 Overview of the Android OS

Android is a Linux based Operating System that runs on a large quantity of mobile devices. Primarily it is installed on mobile phones and tablets, although there have also been cases of the operating system being run on televisions and games consoles as well. Android is currently the most popular OS in the smartphone industry with an estimated 70% of all smartphones sold in 2012 having Android installed on them [3].

Android is an open source operating system that is maintained by Google. This makes it very easy to develop applications for Android because there is a large volume of documentation and many developer tools available.

Because of its accessibility and popularity, Android was chosen as the target platform for this project.

1.5.1 Android Versions

Android has a number of different versions that were developed with different devices in mind. Currently version 2.3, nicknamed Gingerbread, is the most widely used version built for mobile phones. It is installed on 44% of all Android devices. Version 4.2, nicknamed Jelly Bean, is the latest version of Android and it was developed specifically for tablets and high end powerful phones. It currently is installed on 16% of all Android devices [4]. Because of its popularity it was decided that the minimum version of Android that this project would be developed on would be Gingerbread. This means that this application would not be available to any device running an older version of Android. Currently the amount of devices this affects is under 10%.

Android allows you to develop for all or a specified selection of its versions. Different features are added or removed with each new version. How it deals with the differences depends on the specific version. To get around this the Android Manifest file, an XML file that defines essential information about the application, allows you to specify the minimum Android version you are developing for. This means that your project will not be built for any of the versions that came before this minimum version. This allows developers to utilise the more recent features of the Android OS without having to worry about those features not existing in previous versions.
1.6 Libraries Used

Rather than programming everything from scratch it was decided to utilise a number of freely available algorithms and libraries for this project. This saved development time but also produced some challenges when the features required were not an exact fit.

1.6.1 OpenCV

OpenCV (Open Source Computer Vision Library) is the most widely used library for computer vision. Developed by Intel, it is designed to assist programmers working with real time image processing problems. Originally it was written for C++, but it has since been ported to Android to allow mobile developers to access the tools necessary to work with computer vision [5].

1.6.2 Tesseract

Tesseract is an open source optical character recognition (OCR) engine designed for various operating systems. It was originally developed as closed source software between 1985 and 1995 by Hewlett-Packard (HP). HP released the code as open source during 2005. Google have since sponsored development of the Tesseract project and it is currently considered to be one of the most accurate free OCR engines available [6].

Tesseract was not written for Android. Instead this project utilises a library designed by Robert Theis that allows the Tesseract code to run natively within the Android environment [7].
Chapter 2: Background

The image processing that is performed in this project is achieved using the OpenCV library. This chapter provides an overview of some of the main image processing methods that are employed in the project, the theory behind them and documentation on how to perform these methods using functions that are built into OpenCV.

Please note that a “Mat” is an OpenCV matrix object that is used to store all of the information from an image to make it easier to manipulate. A single channel Mat is a greyscale image and a 3 channel Mat is an RGB image.

Below is a table showing the main image manipulation tasks used in this project and the theory that was used to perform these tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Approach Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find Sudoku</td>
<td>Edge Detection</td>
</tr>
<tr>
<td></td>
<td>Contours</td>
</tr>
<tr>
<td>Separate Sudoku</td>
<td>Contours</td>
</tr>
<tr>
<td></td>
<td>Hough Lines</td>
</tr>
<tr>
<td></td>
<td>Warp Perspective</td>
</tr>
<tr>
<td>Detect Number</td>
<td>Contours</td>
</tr>
<tr>
<td>Recognise Number</td>
<td>Tesseract API</td>
</tr>
</tbody>
</table>

2.1 Edge Detection

Edge detection is an essential image processing tool. It segregates the changes in properties within an image allowing much greater accuracy to be achieved in processes such as feature detection by only highlighting the points where there is a sudden change in the image.

Edge detection is defined as the process of identifying and highlighting the discontinuities in an image. The points at which the brightness changes suddenly are the points that are termed to be the edges within the image. This is achieved by measuring the intensity change in brightness between adjacent pixels. These discontinuities can occur due to
changes in depth, variations in surface orientation, material properties changes or differences in scene illumination. Edges have a rate of change (gradient) and orientation.

First derivative edge detectors typically calculate the gradient of the edge as the Root-Mean-Square of a combination of two orthogonal partial derivatives:

$$\nabla f(i, j) = \sqrt{\left(\frac{\delta f(i, j)}{\delta i}\right)^2 + \left(\frac{\delta f(i, j)}{\delta j}\right)^2}$$

They also use a combination of the orthogonal partial derivatives to calculate the orientation of the edge:

$$\phi(i, j) = \text{atan2}\left(\frac{\delta f(i, j)}{\delta j}, \frac{\delta f(i, j)}{\delta i}\right)$$

Second derivative edge detectors look at the rate of change of the gradient of the edge. The most common second derivative filter is the Laplacian filter. The Laplacian filter looks at each of the pixels surrounding the centre pixel and gives each a weighting when computing their second derivative. The weightings are shown below:

$$h(i, j) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad h(i, j) = \begin{bmatrix} 2 & -1 & 2 \\ -1 & -4 & -1 \\ 2 & -1 & 2 \end{bmatrix}$$

Because of this weighting the Laplacian filter is subject to noise so some smoothing is normally performed on the image before using the filter.

### 2.1.1 Canny Edge Detector

The Canny edge detector utilises a combination of the first and second derivative filters to calculate the edge gradient and orientation within an image. The three main goals it was designed to achieve are:

- Low error rate – Good detection of all edges
- Localisation – Edge pixels should be as close to actual edge location as possible
- Minimal response – Each edge should only have one detector response
To reduce noise, first the canny detector convolves the image with a Gaussian mask with an inputted value of standard deviation, $\sigma$. The first derivative is then calculated to estimate the orientation of the edge. The edges are then found by searching for the points where the second derivative crosses the x-axis, as shown in Figure 2. Non-maxima suppression is applied to these edges. Non-maxima suppression ensures that each edge is only 1 pixel wide by omitting edges that are adjacent to stronger ones. The edge gradient is calculated based on the first derivative calculations. Thresholding is then used to find the definitive edge points. If the point is above the upper threshold line then it is said to be the edge point. A lower thresholding value is used to find the edge points that are connected to this edge point as most edges will be more than a single pixel in length. Any point within the two thresholds is said to be a connected edge pixel.

Figure 2 – Overview of Canny Edge Detection Algorithm [8]

The OpenCV Canny function takes in a greyscale Mat and outputs the edges found within that Mat into an output Mat of the same size as the input Mat.
2.2 Contours

A contour, in computer vision, is defined as a list of points that represent a curve in an image [8]. This means that all of the connected points in an image can be defined in a single contour. This is a very simplified overview of what a contour is. There are other properties of a contour that provide more information about the nature of the contour such as whether it is an outer edge or a hole within a contour. However for this project all that is needed is to find all of the connecting points so the above definition is suitable.

OpenCV provides a function to find the contours in an image. This function takes in a binary image and returns the contours in various forms. The return form used in this project was `CV_RETR_LIST` which returns all of the contours found by the function in no particular order. Each contour is returned in a vector containing all of the points in that particular contour. These vectors are all stored in another vector making isolating each individual contour quite simple once you understand the vector structure. The function also offers to reduce the amount of points returned in each contour. It can compress the continuous horizontal, vertical or diagonal segments and just store their end points instead of every point within the contour.

![Image of contours](image)

Figure 3 - Normal Image (left), Edge Detected (centre), Contours Highlighted (right)
2.3 **Hough Lines**

The Hough transform is a method for finding shapes within an image, such as lines or circles. The theory behind the Hough transform is that any point in a binary image could potentially be part of a set of possible lines. The Hough transform maps every potential line into a matrix called the Hough Space. For every point in an image there are an infinite number of lines that could potentially intersect at that point. The Hough transform maps each of the non-zero pixels in a binary image into the Hough Space. Each of these pixels will be represented in the Hough Space by a faint Sine wave that represents every line which could intersect at that pixel within 360 degrees. The Hough algorithm then looks at the local maxima within the Hough space. Where there is a high concentration of overlapping Sine waves, it is said that the points those Sine waves represent are on the same line.

Figure 4 shows a crude drawing of a square and its resulting Hough Space. There are 10 distinct overlaps that can be seen to be much more concentrated than any other overlaps within the Hough space. Figure 5 colour codes each of these local maximum intersects with the corresponding line that it represents in the image. The reason that there are 10 local maxima representing only 4 lines is because each line has 2 orientations in 360 degree space. This would mean every line has 2 local maxima each. Vertical lines lie on the cut off point between 0 degrees and 360 degrees meaning vertical lines will have 3 local maxima each, one in the centre and one at either edge.

```cpp
void findContours(const Mat& image, vector<vector<Point>> &contours, int mode, int method)

- image – Single-channel 8-bit input image
- contours – The detected contours. Each contour is stored in a vector of points
- mode – The contour retrieval mode
- method – The contour point compression method
```
Figure 4 - Picture of a square and its resulting Hough Space (Colours inverted for clarity)

Figure 5 - Local maxima highlighted in Hough Space with their corresponding lines [11]
OpenCV provides a function that computes the Hough algorithm and returns a vector containing all of the lines found in the binary image.

```
void HoughLines(Mat& image, vector<Vec2f>& lines, double rho, double theta, int threshold, double srn=0, double stn=0)
```

- image – Single-channel 8-bit input image
- lines – The output vector of lines
- rho – Distance resolution of the accumulator in pixels
- theta – Angle resolution of the accumulator in pixels
- threshold – If above threshold there is a line present
- srn – The minimum line length
- stn – The minimum gap between each parallel line

2.4 Perspective Transformation

Every photo taken is subject to certain distortions. Lens distortion is caused by the curvature of the camera lens and results in straight lines being slightly curved in an image. Photos are also subject to a distortion due to the angle at which the photo is taken. If the camera is not exactly 90 degrees to the object, identical lines can appear to be different lengths and angles because of the camera’s perspective.

To counter these issues OpenCV provides a `warpPerspective()` function that transforms every point in the image to another image based on a transformation matrix. OpenCV also provides another function, `getPerspectiveTransform()`, which calculates the transformation matrix when given 4 points in the image and the 4 points that these points should map to in the output image. The best points to map with this method are the 4 corner points of the object, in the case of this project, the Sudoku. An illustration of how these two functions work together is shown in Figure 6.
void warpPerspective(const Mat& src, Mat& dst, const Mat& M, Size dsize, int flags=INTER_LINEAR, int borderMode=BORDER_CONSTANT, const Scalar& borderValue=Scalar())

- src – Source image
- dst – Destination image; will have size dsize and the same type as src
- M – $3 \times 3$ transformation matrix
- dsize – Size of the destination image
- flags – A combination of interpolation methods, and the optional flag WARP_INVERSE_MAP that means that M is the inverse transformation (dst → src)
- borderMode – The pixel extrapolation method.
- borderValue – A value used in case of a constant border. By default it is 0
2.5 Optical Character Recognition

The Tesseract library [14] was designed to take an image of an entire sheet of paper and return all of the text within the page. It is a very comprehensive OCR engine that was designed to recognise words and not just the characters within an image. However this project only uses a small fraction of its functionality so a brief overview of how it works is all that will be provided.

Tesseract uses a number of complex methods in order to recognise the words in an image. Firstly it performs a connected components analysis on the image and stores all of the outlines of the characters. This is a similar method to performing edge detection and then finding the contours of these edges, as explained earlier. Tesseract then computes algorithms to find the lines of text in an image. As it can find each text line in an image, it negates the need for Tesseract to warp the image in any way. Instead, it takes into account that the characters may be rotated when matching them to the trained data. Text lines are then broken into words by examining the character spacing within the lines. A large gap will be classified as a space and groups of characters with small gaps between them are defined as words. These words are then passed to an adaptive classifier that attempts to recognise the words when compared to the training data. It also uses an English dictionary to give a higher weighting to more commonly used words. Because the classifier is adaptive, it learns from each word that is inputted into it. In order for Tesseract to fully use any information it gained from the entire image, it performs a second pass to the adaptive classifier that may produce better results for some of the words recognised at the start of the first pass.

This is a very broad overview of how the Tesseract engine recognises words. A lot of its functionality is unnecessary when it is applied to this project’s requirements. The OCR engine is only required, in this application, to recognise an image containing a single character. Therefore the text line computation and the common words aspect of the process are not used. However, due to the complexity of the library, no attempt was made to remove this functionality from the recognition phase of the application. This introduced a performance impact by extracting excessive characters which are invalid in the context of a Sudoku such as letters or false positive numbers. This was addressed by ignoring such results as described in section 3.3 of this report. Due to the lack of
alternative open source OCR engines available, it was decided to still use Tesseract despite these issues.

To use the Tesseract API it must first be initialised. This involves telling the library what language it is trying to recognise and loading the training data provided with the library into it. Then any necessary variables need to be set. In this case the application is using a Whitelist to only search for the relevant characters (the numbers 1 to 9). Finally a Bitmap image is loaded in. The recognised text is then returned as a string and the confidence factor is returned as an integer.

```java
//Set variables for tesseract API
TessBaseAPI baseApi = new TessBaseAPI();
baseApi.setDebug(true);
baseApi.init(DATA_PATH, lang);
baseApi.setVariable(TessBaseAPI.VAR_CHAR_WHITELIST,"123456789");
baseApi.setImage(image);

//Get text and confidence factor
String recognizedText = baseApi.getUTF8Text();
int confidence[] = baseApi.wordConfidences();

baseApi.end();
```

The training data file was provided preconfigured with the library. It contains data to match every character in the English language in a number of different fonts along with a dictionary of common words. Because of this the whitelist is used to just look for numbers and ignore all of the other characters. It is possible to train the library separately but this requires a large amount of time and effort and it was decided to not perform the specialised training for this project in order to allow more time to be used in solving the other issues within the project.
Chapter 3: Implementation

This chapter will discuss how the theory in chapter 2 was utilised to perform each of the major steps required by the background processes of this application.

3.1 Step 1: Capture Photo

In order to capture an image to process it was decided to utilise the native camera in the Android OS. To do this it was necessary to use Android Intents. An Intent in Android is a method to communicate with a background process or service. The filepath where the image was to be saved was specified in the Intent and when the camera was selected an Intent was sent which triggered the capture of the image.
image is to be stored and the image naming convention needs to be set before calling the image capture intent. This then launches the camera and stores the resulting photo in the fields set earlier.

```java
protected void startCameraActivity()
{
    //get path to store image at
    String path = Environment.getExternalStorageDirectory() + "\WhereDidIGoWrong/CurrentSudoku.jpg";
    File file = new File(path);
    //Create output file
    Uri outputFileUri = Uri.fromFile(file);
    //Start camera intent
    Intent intent = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
    //Save resulting image in output file
    intent.putExtra(MediaStore.EXTRA_OUTPUT, outputFileUri);
    startActivityForResult(intent, TAKE_PHOTO);
}
```

This image then needs to be sent from the PhotoCapture Activity to the ImageManipulations Activity. An Intent is again used to achieve this. This time the Intent triggers the ImageManipulations Activity to start and also sends the filepath for the desired image at the same time. The ImageManipulations Activity can then open the image from the filepath that it received and start to process the image.

![Figure 8 - Application Code Structure](image-url)
3.2 Step 2: Sudoku Extraction

To extract a Sudoku from an image first it is necessary to define to the programme what exactly a Sudoku is. Every Sudoku contains a 9 x 9 grid of boxes, some containing numbers, while others are empty. Each Sudoku is square in shape and should not have anything touching its outer edges. In the vast majority of cases a Sudoku will be printed on a light sheet of paper with a dark shade of ink.

These definitions are necessary in order to successfully isolate the Sudoku from the rest of the image. This allows the Sudoku to be processed on its own, so resources are not wasted trying to find the Sudoku in the scene every time the Sudoku is referred to.

3.2.1 Edge Detection

Because there is a large contrast between the well defined, dark lines of the Sudoku and the light background, edge detection was the method chosen to isolate the necessary details and remove some of the noise from the image. The bitmap of the Sudoku was converted to an OpenCV Mat and this Mat was then converted to a greyscale Mat using the cvtColor() function. The greyscale Mat was then passed into the Canny() function. This produced a binary image of the original photo with all of the edges highlighted in white and everything else black as shown in Figure 9.

![Figure 9 - RGB image and resulting edge image produced using Canny() function](image)

3.2.2 Contour Detection – Find Border of Sudoku

Once an edge image has been created the Sudoku within it needs to be isolated. There are a number of different ways this can achieved. Finding all of the lines in the image using the HoughLines() function was initially thought to be the best method to find the Sudoku. However upon implementation, it was found that there was no easy definition for what the Sudoku actually was. This method resulted in a lot of lines highlighted within...
the scene but no clear way of defining which lines were from the Sudoku in question and which were unnecessary lines, as shown in Figure 10.

![Figure 10 - Houghlines() on entire edge image](image)

It was decided that an assumption would need to be made at this point. This assumption was that the Sudoku would need to be the main focus of the picture that needed processing. This was a logical assumption, as the Sudoku is the focus of the application.

To extract the largest object from the image it was necessary to look at the contours within the picture. A contour in computer vision is defined as a continuous chain of matching pixel values. The `findContours()` function takes in the edge image that was created earlier and scans for these connecting contours throughout the image. Each individual contour is stored as a vector of points. It is then possible to manipulate each individual contour to extract details, such as the length of the contour, or to display the contour on the image using the `drawContours()` function.

Once the contours were found it was necessary to iterate through each of them to discover which was the largest. To do this the `contourArea()` function was utilised. This function returns the area in pixels that is within the specified contour. Once the largest contour was found it was said to be the contour that surrounded the outer limits of the Sudoku. This contour may not necessarily be a perfect square due to camera lens distortion, and the angle at which the photo was taken. To resolve this, a bounding rectangle was placed around the extremities of the contour and then copied to a new
Mat. This extracted an imperfect Sudoku but provided a smaller Mat containing only the Sudoku to work with.

3.2.3 Correcting Sudoku Distortion

The extracted Sudoku was not likely to be perfect. It was subject to various distortions that, unless dealt with, made dividing it into the required 9 x 9 cell arrangement very difficult. These distortions could come from the various lens distortions of the camera used (Figure 12), or even the angle at which the photo was taken.

As is displayed in Figure 13, if the photo was rotated slightly, the rectangle used to bound the Sudoku would also include a lot of unwanted extra space, and the Sudoku itself would not be straight. This was dealt with in two steps, by first locating the corner points of the Sudoku within the image, and then transforming the entire image to a square image.

The first step used HoughLines() to locate all the lines within the image. A constraint was placed on the lines that their length had to be a minimum of half the image width or height. This found all the grid lines of the Sudoku, as shown previously in Figure 10. These lines were then iterated through and the line that was closest to each edge of the image was stored as highlighted in Figure 13. The intersection of each of these lines was then calculated using the formula:
Each intersection point was taken as a corner point of the Sudoku and stored to use later.

The corner points of the Sudoku were put into an array of points in a specific order (top left clockwise to bottom left). Another array of points was created, containing the corner points of a Mat of the same size as the extracted Sudoku Mat, in the same order as above. The `getPerspectiveTransform()` function was used to find the transformation matrix between the two Mats. This was then used in the `warpPerspective()` function resulting in a corrected Sudoku image being produced as shown in Figure 13.

![Figure 13 - Angled Sudoku extracted (left), Outer lines highlighted (centre), Perspective warped (right)](image)

### 3.3 Step 3: Extracting Printed Numbers

Once the Sudoku was warped into a square Mat containing just the Sudoku, the next step was to recognise the numbers. To do this an assumption that the Sudoku has been warped correctly is made. This meant that the grid lines of the Sudoku should be within five degrees of 0 or 90 depending if they are horizontal or vertical.
The first test performed to extract the numbers was to see if the OCR engine would be able to recognise the numbers in the Sudoku when presented with the entire image of the Sudoku. The grid lines were removed from the Mat by using the Hough Lines algorithm again to show all the lines that were longer than half the size of the Mat. These were then removed by drawing white pixels over the line pixels. This image was then sent to the OCR engine. The test failed because the OCR engine did not return any details about where the characters were found in the image. All that it provided was a long string of characters that it had found within the image.

It was then decided to segment the image into a 9 x 9 array of cells and pass each of these cells to the OCR engine individually. This was achieved by dividing the image width and height by nine. Then a rectangular region of interest submat was put around each box and these were stored in a 9 x 9 two dimensional Mat array. The gridlines added some noise to each of the submats. To remove the gridlines, each submat was reduced in size by a pre-set scaling factor in every direction. A good scaling factor was found to be the width of the Sudoku Mat divided by 50.

![Figure 14 - Image segmented into cells (left), Each cell recognised with no constraints (right)](image)

The next test was to send these submats to the OCR engine individually and to print the resulting characters on each of them. This is shown above in Figure 14. It is obvious from the image that there are a lot of false positives being returned by the OCR containing incorrect characters that will never appear in a Sudoku. To solve this problem, the OCR
engine was initialised to only look for numbers when it was given an image as shown previously.

```java
gbaseApi.setVariable(TessBaseAPI.VAR_CHAR_WHITELIST,"123456789");
```

This resulted in an improvement as the OCR was no longer returning the garbage characters present in Figure 14. However it did not solve the issue of the false positives in cells that did not contain a number. It was decided to try and only send cells that contained a number to the OCR engine. To do this some processing needed to be performed on each cell before it was sent to the OCR engine. The `numberPresent()` function takes in a Mat containing the cell that is to be checked and returns true if there is a number in it or false otherwise. To judge if there is a number in a cell the function uses `Canny()` to get an edge image and `findContours()` to find all of the contours within that image. If the area taken up by these contours was more than a certain threshold then it was said that there was a number within the box and the function returned true. The cells that contained numbers were then sent to the OCR engine and those without numbers were not. The results are shown below in Figure 15.

![Figure 15 - Only recognising numbers (left), Only recognising cells with a number in them (right)](image)

This was a vast improvement with 80% of the numbers being recognised correctly and the OCR engine no longer returning any false positives. However the results still were not perfect. There was still noise in the images being sent to the OCR engine. The easiest way to remove this noise was to send only the number to the OCR and remove any extra pixels surrounding it. To implement this, all that needed to be added was a method to
take look at the largest contour in each of the cells that had been identified as containing a number. This function took the largest contour within each of these cells and put a rectangle around the extremeties of the contour plus one pixel in each direction. It then returned just the contents of this rectangle as a new Mat. This focused image of just the number was sent to the OCR engine and provided the best results with over 95% accuracy on printed numbers.

![Image](image.png)

**Figure 16 - Individual Cell (left), Largest contour in cell highlighted by rectangle (centre), Result when largest contours recognised (right)**

### 3.3.1 Manually Correcting Recognition Errors

The number recognition method worked for 95% of printed numbers but sometimes there were numbers that it interpreted incorrectly. This could be due to smudges on the printed image, a poor quality photo or noise in the image. Because of this, functionality was included in the application to manually set every number within the Sudoku, thus overriding incorrectly recognised cells. This is achieved by positioning an invisible button behind each cell. This will be discussed in the User Interface section of this report. Once a button is pressed the user will be given the option to choose a number between 0 and 9 to put in the cell. 0 will set the cell back to having nothing recognised in it.

### 3.4 Step 4: Solve Sudoku

The algorithm used to solve the Sudoku in the project was adapted from an applet published online that solves the puzzle using recursion [15]. Because a Sudoku is purely logical, in theory, a person trying to solve the puzzle should never have to make a guess as to which number goes in a certain cell. There should always be a way, through logic, to
deduce at least one number in the puzzle which should then lead to another number and so forth.

There are many algorithms online that try to implement all of the logical options that a human will be able to deduce in order to solve a Sudoku. However these can be very long and complex. For this project it was deemed acceptable to utilise a simple brute force algorithm in order to get a solution.

The algorithm chosen is a demonstration of recursive programming. It is a very simple method that repeatedly calls itself until a solution is found or the algorithm reaches the last cell in the puzzle in which case the puzzle could not be solved. It works by searching the puzzle from the top left corner, to the bottom right, moving one cell at a time. When it finds a cell with no number in it, the algorithm puts a 1 into this cell and then checks if this causes a clash with another 1 in the same row, column or box. If it does not cause a clash, the algorithm moves to the next empty cell in the row and puts a 2 into it and again checks each row, column and box for a clash.

When a clash is triggered the algorithm breaks out of its current solve call and moves back one step. It then tries to put the next number along into the box and continues as normal. This process continues until there are no empty cells left or if the algorithm reaches the last cell and cannot put a number into it.

It should be noted that this is a relatively inefficient method to solve a Sudoku. Testing showed that it makes on average 40,000 recursive calls to the solve function when it successfully solves a medium difficulty level puzzle. When it failed to find a solution the average number of recursive calls to the solve function was 20 million. However, due to the powerful capabilities of modern smartphones, the devices this method was tested on took under a second to complete the algorithm every time. Therefore it was decided that it was an acceptable method to solve the problem, given that it is guaranteed to solve any Sudoku that has a solution.
3.5 Step 5: Dealing with Handwriting

The OCR engine does not just recognise characters. It can also provide a factor of how confident it is that it has correctly identified a number. This factor, or “Quality Metric”, was used to help differentiate between printed numbers and handwritten ones. The method for sending the numbers to the OCR engine remained the same. However, when the recognised character was put into the Sudoku cell array, the quality metric was stored with it as an integer. It was found through testing that a quality metric of 80% was a good threshold to differentiate between handwritten and printed numbers. Because of the uniformity of the printed numbers, their quality metric scores were usually in the 90s. The handwritten numbers tended to produce much lower confidence scores. To display the difference between the two types of identification the application writes the printed numbers in green and the handwritten ones in yellow as shown in Figure 17.

Once the printed numbers and handwritten ones can be differentiated it is possible to implement checks to see if the user has made a mistake anywhere within the puzzle. First the application runs the solving algorithm with all the numbers in place and informs the user if it was able to solve the Sudoku. If the solving algorithm fails then the user can “remove” the numbers that have been classified as handwritten and run the solving algorithm again. If the numbers have been classified correctly then the algorithm should be able to solve any solvable Sudoku. The solved numbers can then be compared with
the numbers recognised in the same cells. If they match, the numbers were written correctly. If they are different, the numbers that the user wrote down may have been wrong. The application highlights the wrong numbers, as shown in Figure 17, which allows the user to check if they were actually incorrect or if the OCR engine has misclassified a handwritten number. This misclassification can happen quite easily depending on the handwriting of the user.

3.6 Step 6: Display Numbers
Once the Sudoku has been solved the user is given the choice of how they want to have the results displayed. The errors will be highlighted immediately as mentioned above. However in order to allow the user to continue working on the puzzle, the full solution is not immediately displayed. Instead the user is given the option to choose if they would like a hint or if they want the complete solution. Pressing the hint button will randomly select one of the unrecognised cells and display the solved number in it in pink. Pressing the complete button will fill in the remaining empty cells with the solved numbers in blue.

Figure 18 - Sudoku with Hints Displayed (left) Completed Solution (right)
Chapter 4: User Interface (UI)

The user interface for this application was kept as simple as possible. The native Android UI options were implemented where possible in order for more time to be spent focusing on the back end design of the application. For both activities the UI consisted of:

- Main Imageview - to display the picture that was being processed.
- Multiple buttons - to implement the various functions within the application.
- Small Imageview - to display the status of the application process.

All of the UI was defined in the layout XML files. It was designed to stretch or shrink to various screen sizes in order to work on multiple Android devices.

For simplicity the UI is locked in the portrait view only. In future versions of the application, a layout could be designed to allow the application to run in horizontal mode as well.

![Figure 19 - Overview of UI](image_url)
4.1 Imageview

The main Imageview depicts the image that is being manipulated. It is set to occupy two thirds of the screen which, on most devices, will allow the extracted Sudoku to be viewed as a square that fits the width of the screen. The image that is set within an Imageview will automatically stretch to either the height or the width of the Imageview but will maintain its original aspect ratio. This is why there is a slight gap between the Sudoku in Figure 19 and the top of the screen. Depending on the device screen dimensions this gap will be change size or disappear.

4.2 Buttons and handlers

The applications functionality is triggered by the various buttons displayed on the screen. In the photo taking activity the buttons allow the user to take a photo, select a photo from the gallery, load the sample Sudoku stored in the applications assets folder or process the photo that is currently displayed within the Imageview.

In the image manipulations activity, the buttons allow the user to extract the Sudoku from the original image, recognise the numbers of the Sudoku, remove the low quality numbers, solve the Sudoku and then display the results as hints or display the completed Sudoku.

![Figure 20 - Buttons for both activities](image)

4.3 Invisible Buttons used to select individual cells

In order to add the option to change each number in the Sudoku individually it was necessary to position an invisible button over every cell in the displayed Sudoku image. The other buttons in the application were all defined and positioned in the layout XML document. They then had click listeners applied to them once they were created in the ImageManipulations.java file. The invisible buttons however were created dynamically
within the java file. This led to issues positioning the invisible buttons over the correct numbers when trying to develop for multiple screen sizes.

The biggest issue to overcome was the fact that Android does not have the functionality to return the dimensions of an image in an Imageview if that image was loaded after the Imageview was defined. The Sudoku image was scaled to fill the Imageview every time it was created. However the Imageview was defined in the `onCreate()` stage of the ImageManipulations Activity. Android provides the functionality to get the Imageview’s dimensions and to get the actual dimensions of the image within the Imageview but it does not provide the scaled image’s dimensions. Therefore, it was necessary to calculate these.

![Image Diagram](image)

*Figure 21 - Calculations required to get scaled image size*

To do this, the aspect ratio of the actual image was calculated by calculating the tanTheta of the height/width as shown in the code below.

```java
double tanTHETA = intrinsicHeight/intrinsicWidth;
double correctedHeight = tanTHETA * imageViewWidth;
```

This was then multiplied by the width of the Imageview to get the corrected height of the scaled image. If the corrected height was greater than the Imageview height then the
scaled image’s width needed to be changed instead. The same process was implemented with the heights and widths reversed.

\[
\tan\theta = \frac{\text{intrinsicWidth}}{\text{intrinsicHeight}}; \\
double \text{correctedWidth} = \tan\theta \times \text{imageViewHeight};
\]

Once the scaled image size was known, the button sizes could be calculated by dividing the image size by 9. The buttons were then dynamically created in a 9 x 9 array within a TableRow that was defined in the layout XML file as having the same position and size as the Imageview. Each of the buttons was assigned an ID number that equaled 1200 + (the row the button was in * 9) + the column that the button was in. These ID numbers were used to work out which button was pressed within the button click handler. Once a button is pressed the user is prompted to insert the number for that cell into it.

![Figure 22 - Buttons scaling correctly to Sudoku cells (shown as visible for demonstration purposes)](image-url)
4.4 Progress Bar

The number recognition performed by the Tesseract API on average took about 5 seconds to complete. On less powerful devices it could take up to 10 seconds to cycle through every box with a number and have that number processed and recognised. This made the application look like it was doing nothing when the Recognise button was pressed. To deal with this issue it was decided to add a Progress Bar to indicate that the application was still functioning correctly. To implement the Progress Bar it was necessary to thread the application. The actual processing of the image was sent to a background thread and the Progress Bar was displayed until this thread had finished recognising all of the numbers. This lead to a bug when the Recognise button was pressed and the Sudoku had not been extracted from the original image. The progress bar would appear but never be dismissed because the application would not try to recognise the numbers without the extracted Sudoku. To counter this, a check was put in place to see if the Sudoku had been extracted before the progress bar was called. Also the progress bar was set so that it could be dismissed if the back button was pressed on the device. This prevented the progress bar from crashing the application.

![Figure 23 - Progress bar showing background activity being performed](image-url)
4.5 Toasts

Once the solving algorithm had been run the user needed to be informed if the Sudoku had been solved or not. To do this, the Android Toast feature was implemented. A Toast is a small dialogue that appears for a few seconds that contains information for the user. They are relatively simple to implement.

![Toasts indicating result of solving algorithm](image)

4.6 Status Indicator

The Status Indicator at the bottom of the UI allows the user to quickly see where they currently are within the application. It helps them to see what still needs to be done before they can view the solved Sudoku. The Status Indicator is another ImageView containing the current status image. The current status images were created using Paint.NET and are included within the application as assets. There are different versions of each image created for different resolution screens in order for the text to be visible at lower resolutions. Whenever the application completes a key milestone the Status Image is updated to indicate what has been completed.

![Status Indicators](image)
Figure 26 - UI of ImageManipulations activity shown on a device
Chapter 5: Testing

Throughout this project a vast amount of testing was executed to ensure every aspect of the project performed as expected. In order to check each new method that was introduced to the implementation, a test UI was used extensively. The test UI contained buttons to check each algorithm in turn as well as functionality to zoom into specific numbers of the Sudoku in order to examine issues that cropped up when sending the numbers to the OCR engine.

A number of smaller applications were also created to individually test certain features within the main application. These small applications included:

- A photo taking application
- An application to send images to different activities using Intents
- An application to send a specific image to the OCR engine and show the results

Once the functionality was fully understood and implemented correctly, it was then copied into the main application and again tested extensively.

The main UI was also tested comprehensively to see how it scaled to differing sized screens. It was designed so that the buttons and Imageviews all scaled to any sized screen, but there were a number of issues with buttons appearing incorrectly on very small screens. These problems were minimised by using a cheap small Android phone when testing and ensuring that the UI appeared correctly on it.

The majority of implementation and UI tests were performed on two specific devices:

- The Sony Xperia U smartphone – Android version 2.3.3
- The Google Nexus 7 tablet – Android version 4.2

These are both middle of the range Android devices. Therefore they were a good indicator when it came to the performance of the application.

Every time the number extraction methods were changed, the application was tested against a test set of 40 Sudokus. These 40 photos were taken of Sudokus with no handwritten characters and were taken from 3 different newspapers: The Irish Times,
The Irish Independent and The Metro Herald, as well as from Sudokus printed from online sources. The test set of photos used are attached as an appendix to this project. The test involved opening each image, extracting the Sudoku, recognising the numbers and then solving the Sudoku. An image passed the test by managing to solve the Sudoku. The results for each number recognition method are displayed in Figure 27.

![Figure 27 - Results of number recognition method tests](image-url)
Chapter 6: Bringing to Market

Once UI scaling issues were solved and the applications functionality thoroughly tested, the next step in this project was to release the application to the public. The central repository for the majority of Android applications is called the Google Play Store; the latest figures show that there are over 600,000 applications available on it [16]. Every Android device that can connect to the web has access to the Play Store, provided that the device has been registered and paired with a Google Mail account.

Releasing an application on the Play Store is a simple process. A programmer needs to sign up to Google’s developer website, agree to the terms and conditions and pay a once off fee of $25. Once this has been completed, the developer is free to publish as many applications as they want provided that they do not breach the terms of service. Google has options for the developer to set up a merchant account if they wish to upload paid applications and there are also options to select which countries the application will be distributed in.

Applications are installed on Android devices using the application package file type (APK). To upload an application to the Play Store, the developer only needs to upload the APK file of the application. The APK file packages the applications code, resources, assets, certificates and the AndroidManifest together.

The development platform used to build and test Android applications creates a debug APK file every time it builds the application. However Google requires that all APKs uploaded to the Play Store must be a release version APK. A release version APK file needs to be created separately. This file has more structure and rules defining it than a build APK. The biggest difference between release and build APKs is that release versions are signed with the developers own certificate in order to encrypt the data within it. A release APK must also have a unique java package name defined for this specific application. These package names cannot be reused. When creating an application in Eclipse, the developer is prompted to name this java package

com.example.yourapplicationname. In order to create a release APK this must be changed to a unique name. In the case of this project it is
“com.millerg.wheredidigowrong”. A unique version code for the application must also be defined within the AndroidManifest file. Once the application has the unique package name, version code and is signed with the developer’s certificate, a release version of the APK can be built using Eclipse’s tools to export a signed APK.

This APK file is then uploaded to the Google Play Store and the developer is asked to then input some information about the application and some screenshots in order for the end user to be able to find and choose this application to install. The application will then be published online within a few hours and available for anyone to install on their Android device.


### 6.1 Issues

Once the application was uploaded to the Play Store it was then tested by downloading it onto a number of different devices. A number of issues arose with the application crashing immediately every time it was run on a new device. Examining the logs of these crashes revealed that there was an issue with the OpenCV library and the Tesseract library not being found by the device. The problem was that Eclipse automatically configures the build path to reference the chosen libraries for the debug APK file, but
does not do so for the release version. The crashes were fixed by changing the release version build path to reference the two libraries used and then uploading an updated version of the application.

Other issues arose with some Android devices not having cameras to take photos of Sudokus. Also a number of tablet devices only have a front facing camera making it very hard to take a photo of a newspaper because the device needs to be turned backwards. To attempt to address these issues functionality was added to the application to select any photo from the device’s gallery. This was achieved using Android’s native photo picking intents that are built in to the OS. Triggering an intent will change the current activity of the device to the Gallery application. The user will then be able to use this application to select a photo of their choice. Once they have clicked the photo the intent will return the storage path of that photo to the application and this can then be used to load the photo into the Imageview and then sent to the ImageManipulations activity to process the image.

```java
//Triggers Intent to select a photo from the device's gallery
protected void selectPhoto(){
    Intent photoPickerIntent = new Intent(Intent.ACTION_PICK);
    photoPickerIntent.setType("image/*");
    startActivityForResult(photoPickerIntent, SELECT_PHOTO);
}
```

A sample Sudoku was also packaged as an asset with the application. Functionality was added to store this sample photo on the device upon running the application for the first time. This photo could then be loaded in a similar manner to the select photo method by pressing the “Load Sample Sudoku” button.

With these two features added, the Sudoku finding and solving methods of this project could now be demonstrated on any Android device regardless of whether it had functionality to take a photo or not.
Chapter 7: Conclusion

The core goals of this project were successfully achieved with a working application being developed and provided for download from the Google Play Store. This project offers the user a method which can assist them in solving a Sudoku puzzle if they cannot see the next step to take or have made a mistake somewhere in their method. An Android application was developed and published which can take a picture of a printed Sudoku, extract the relevant data, differentiate between printed and handwritten characters, solve the puzzle and then notify the user where they may have made a mistake.

This application differs from other applications available on the Play Store in the fact that it attempts to deal with the presence of handwritten numbers on the puzzle. The other similar applications available only try to solve a “clean” untouched Sudoku which is not a very useful tool for the end user. The purpose of a Sudoku is that it is a puzzle that the user wants to solve. Giving the solution before the puzzle has been attempted completely defeats this purpose. Offering the user the option to try to recognise their handwriting makes this application a great deal more useful to them. The handwriting recognition rate is not perfect, so the user is offered the option to manually input the characters that the OCR engine may have recognised incorrectly or missed. The fact that the solving algorithm only uses the printed values to solve the puzzle allows the application to highlight where the solved puzzle does not match with the handwritten characters.

7.1 Further Development

The current method for extracting the Sudoku is sufficient for this project’s purpose, but the number recognition methods could be improved. Differentiating between handwritten and printed numbers is done relatively crudely by utilising the quality metric of the recognised number. Sometimes handwriting can be neat enough to score a quality metric over 80% in which case the application will think that it is a printed number. However these can still be overridden by the manual correction option.

Other potential methods for differentiation were to constrain the user to using a coloured pen (blue, red or green). In this way the colour of the number could be established and then classified as handwriting, while printed numbers would always be
black. This method was not employed in this project due to the fact that this puts an unreasonable constraint on the user. The objective for this application was for it to be as user friendly as possible. The idea behind it was that the user would only use this application if they became stuck while solving a puzzle. They would not necessarily plan in advance what coloured pen to use because they envisioned themselves making a mistake. For this reason the colour checking was not attempted with this version of the application. It could be included as an optional feature in future versions of the application.

Another method that was considered to help differentiate the handwritten and printed numbers was to look at the thickness of the lines of each character. The idea being that a pen would produce a thinner line than a printed number. An efficient method was not developed in time to test this idea, so it has not been included in this project but some research into this proposed solution could be examined in future versions of this application.

![Figure 29 - Application from start to finish](image-url)
Chapter 8: Bibliography


Chapter 9: Appendix

9.1 Contents of attached CD

- Full source code of this project formatted so that it can be imported into Eclipse
- Current release version of the APK file for this project
- The PowerPoint demonstration of how this project works, as given on 19 March 2013
- A pdf version of this report
- The 40 sample Sudokus used in testing and 9 Sudokus containing handwriting

9.2 Link to the finished application on the Google Play Store


9.3 “Where Did I Go Wrong?” Changelog

30 January 2013 – First Commit to Subversion

- Working photo taking and sending activities
- Provisional button layout and click handlers added

31 January 2013

- Basic OpenCV functionality added. Tested with Canny and RGBA
- Grid and Contour finding added. Working
- Max Contour finding added and working for sample image

1 February 2013

- Max contour sent to submat and displayed

2 February 2013

- Number segmentation into array of matrices implemented and tested. Need to morph image first for best results

3 February 2013

- Number segmentation test tools implemented
6 February 2013

- Functionality to select a sample image from gallery added

9 February 2013

- Functionality to find corners of Sudoku in Sudoku ROI added
- Warp Perspective implemented successfully

10 February 2013

- OCR implementation segmented from main activity. Fixed image displaying issues

11 February 2013

- Contour checking boxes to see whether OCR necessary. Improved performance 50%!!!

12 February 2013

- Added in potential solving functionality

19 February 2013

- Improved number recognition through contours. Still needs work as can’t handle open contours....

20 February 2013

- Improved number recognition with more accurate contour extraction. Works for over 90% of newspaper photos

26 February 2013

- Added functionality to change recognised numbers by clicking on them

27 February 2013

- Added loading widget with threads
- Improved OCR and solving implementations to reduce repetitive work
2 March 2013

- Improved number buttons to match size of images
- Changed colours of displayed numbers

3 March 2013

- Added Status Indicator at bottom of image

4 March 2013

- Added OCR quality detection
- Segmented Sudoku array to its own class
- Added Hint functionality
- Potential bug in solving algorithm. Multiple 3s can be in same box.... Needs proper examination

5 March 2013

- Fixed solving bug
- Fixed invisible buttons sizing issues

6 March 2013

- Added remove functionality to get rid of low confidence recognition

9 March 2013

- Created new project with proper naming conventions and transferred all old data
- Fixed a few bugs in invisible buttons
- Hint and complete now appear if the Sudoku is solved
- First release build uploaded to marker
- Bugs with it crashing on devices when downloaded....

10 March 2013

- Forgot to add tessdata file when transferring from old project. Fixed
- Reworked UI
• Fixed release build path to include external libraries correctly
• Now runs on other devices!

11 March 2013
• Packaged sample image with app and added functionality to load it

12 March 2013
• Updated sample image
• Added multiple testing functionality
• Implemented where I went wrong check. Could be improved

15 March 2013
• Built final version of app for presentation
• Changed number highlighting code to make more user friendly
• Drastic rework of UI to remove all debugging buttons (Still present in old badly named application if needed)

25 March
• Code cleaned up, commented and all debug code removed