Saint-Petersburg State University
Mathematics and Mechanics Faculty

Albina Yezus

Interactive chess viewer

Term paper

Scientific advisers:
Aleksey Artamonov, Software Engineer, Yandex
Viktor Dudin, Software Engineer, Yandex

2015

1 CONTENTS

2 ABSTRACT

In this work, the methods of image recognition and machine learning are used to detect chess diagrams in books and recognize chess pieces on them. Although it may be difficult to come up with the algorithm that works perfectly well in all cases, some steps initial steps towards solution and simplification of the problem are made.

Keywords: image recognition, machine learning

3 INTRODUCTION

This paper is a result of the author's collaboration with Malinovsky Ilya (malinovsky239@yandex.ru). The goal of the work is to create an Android application that allows users to upload chess books and after transformation read them in a format that supports interactive diagram viewing.

The aim of this study is to create a program that transforms chess books into special format, recognizable by Android application. In order to do this it is necessary to solve several problems. Firstly, the income format of the book should be decided as well as format that Android application understands. Secondly, method of diagram detection should be found. The last and most difficult part is piece recognition.

There are several applications on Google Play positioning themselves as "chess book readers". However, none of them provides image recognition or in-book interactive chessboards, thus application to be made in unique in this field.

Several programs that detect diagrams and pieces of them can be found on web as well. However, they work only in specific cases and usually require clear images, which often cannot be found in books due to low scan quality.

4 PROBLEM STATEMENT

To create a program that detects diagrams in books and recognizes pieces there.

Diagram detection involves finding the following:

1. Relative position of the top left corner;
2. Relative width of the diagram;
3. Relative length of the diagram.

Piece recognition involves finding the following:

1. The presence/absence of the piece;
2. Piece color;
3. Chess piece.

5 APPROACH

The entire work can be separated into several steps:

1. Deciding on input and output formats*;
2. Parsing book;
3. Diagram detection;
4. Cells detection;
5. Cell purification;
6. Preparing dataset*;
7. Simple piece recognition;
8. Teaching classifier*;
9. Piece recognition;
10. Embedding into application*.

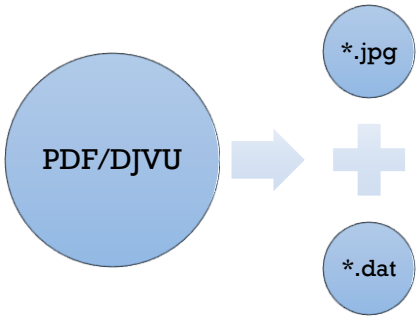
Steps marked by "*" should be done only once, whereas algorithms implemented on other steps must be applied for each uploaded book.

5.1 DECIDING INPUT AND OUTPUT FORMATS

The input format is set to be PDF/DJVU, as they are the most common formats of the chess books.

The problem of the output format is in the fact that this program is going to be built into the Android application. Thus the output format should be as light and easy to access as possible so that it does not take much time to flip over other pages and does not take much memory.

The solution to this problem is depicted bellow:



Information about each page is saved in two files:

1. *.jpg file contains a page image in resolution corresponding user's device;
2. *.dat file contains information about the diagrams and pieces on them:

```
<number of diagrams> '\n'
{<diagram> '\n'}

<number of diagrams> ::= non-negative integer
<diagram> ::= <x> <y> <width> <length> <FEN>
<x> ::= relative x coordinate of the top-left corner
<y> ::= relative y coordinate of the top-left corner
<width> ::= relative width of the diagram
<length> ::= relative length of the diagram
<FEN> ::= first part of the Forsyth-Edwards Notation [1]
```

5.2 PARSING BOOK

After deciding on the output format, parsing books seems straightforward. Now all we have to do is separate book into pages and apply implemented algorithms to each of them.

5.3 DIAGRAM DETECTION

Diagram detection is basically finding square images relative sizes of which are bigger than the given value that needs to be set and tested. Thus this part of the work has involved rectangles detection and their evaluation. The results of this part are relative top-left coordinates of the diagram and the borders' lengths.

5.4 CELLS DETECTION

As it is possible that diagram detection algorithm does not work perfectly, cell detection should exclude images that diagram detector recognized as diagrams and detect cells of the real chessboard images. It is also possible that image of the diagram is smaller/bigger than it actual size, so simple division by eight rows and eight columns is not an option.

5.5 CELL PURIFICATION

The purpose of this part is to clear cells from noise and set them to "almost" uniform form. Unfortunately, we cannot omit the word "almost", as diagrams and pieces are depicted differently in different books, making some standard form impossible to create.

5.6 PREPARING DATASET

Up to this moment, diagram and cell detectors should have already been made. Thus, cell images can be extracted automatically from various books, and their descriptions can be made manually.

5.7 SIMPLE PIECE RECOGNITION

The quality of the purified images can be tested by applying simple algorithms, for example pixel-by-pixel comparison.

5.8 TEACHING CLASSIFIER

In this work, two approaches are going to be made:

1. Histogram of oriented gradients and support vector machine;
2. Artificial neural network

5.9 PIECE RECOGNITION

This part of the work involves feeding cell images to the classifier and storing the result in order to create the output format suitable for the Android application.

5.10 EMBEDDING INTO ANDROID APPLICATION

As the source code is written in Java, it is possible to embed it into Android application. However, due to specification of Android platform, some parts will have to be amended or rewritten altogether. It is also necessary to provide user-friendly interface.

6 ALGORITHMS

The entire book transformation can be divided into several steps. On each steps various algorithms are applied as they are described below.

For the most of the algorithms OpenCV library has been applied (see "Technology" below).

6.1 BOOK PARSING

The following steps can describe the formatting from PDF/DJVU:

6.1.1 Getting a book name from the string path to it

We assume that the name of the book is everything between the last '/' symbol to the last '.' symbol.

6.1.2 Creating directory to store the output format

If such directory already exists, symbol '_' adds to the name of the book and directory with that name is created.

6.1.3 Extracting and saving images in necessary resolution

As the project is not yet embedded into Android application, the resolution of the image sets assuming that the standard height of the screen is 2560 pixels by the following formulas:

$$coef = \frac{2560}{page\_hight};$$

$$img\_hight = 2560;$$

$$img\_width = coef * page\_width$$

6.1.4 Creating a .dat file

For each stored image .dat file is created. The relative coordinates and lengths of the borders for each diagram are easy to find:

$$\frac{\text{absolute\_value}}{\text{image\_value}}$$

In order to create a FEN diagram fast enough the program loops over all cells only once and creates the annotation online. If there is a piece on the cell, its name is added to the FEN string after checking if the counter of the empty cells is 0. If it is more than 0, this number is added to the string before the piece name and the counter is reseted. Otherwise the counter of the empty cells is increased. It is also necessary to reset counter at the end of each row.

6.1.5 Example

.jpg stored image:

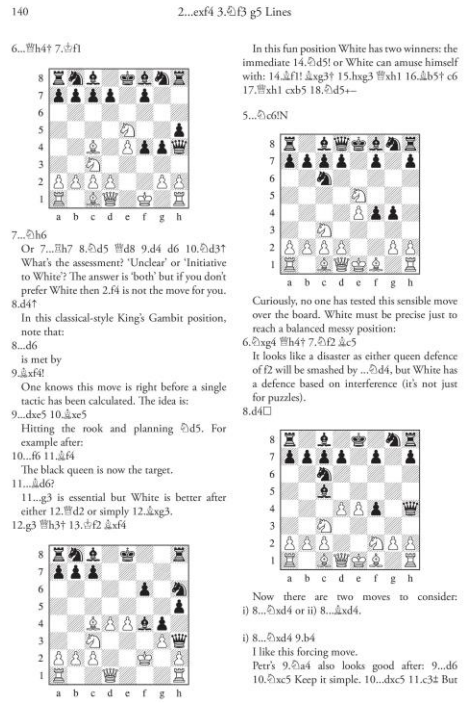


Image example

Corresponding .dat file[1]:

```
4
0.17 0.74 0.25 0.18 rkb1kk1r/bbb1k1k1/3k1b2/2k1k1kb/1kb2bb1/4k2k/3k1k2/r1kkk1kr
0.58 0.59 0.25 0.18 r1b1k1kr/bbbb1bkb/1kkk4/2b1k1k1/5b1k/4k1k1/3k4/r2kkbkr
0.58 0.22 0.25 0.18 r1b1kbkr/rbrbkbkb/1kk2k2/2k3k1/5bb1/b3b1k1/5k2/r1bkkbkr
0.17 0.14 0.25 0.18 rkb1kbkr/bbbb1b2/5k2/6kb/2bk1bb1/4k1k1/5k2/r2kkkkr
```

6.2 DIAGRAM DETECTION

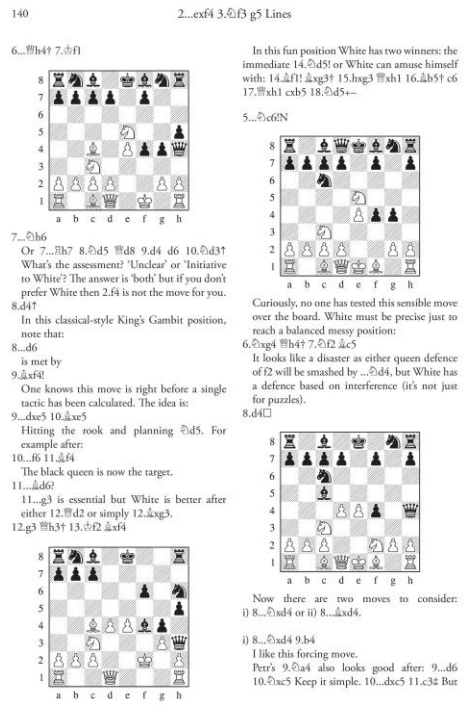
This part of the work describes diagram detection. It appears to work great on finding diagrams and does not miss anything. However, sometimes not only diagrams are detected but other images as well. The method of purging these images is described in the next section.

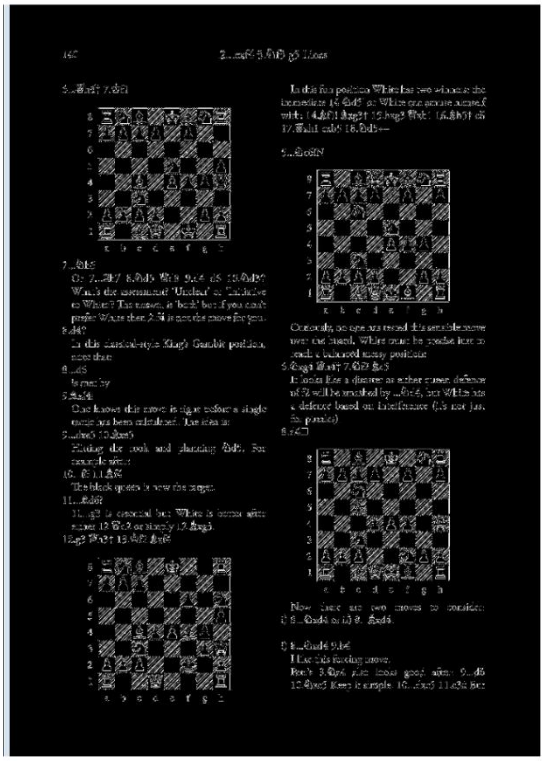
6.2.1 Canny edge detection

The first step is to convert the image to the binary format. The best way to do this is to apply Canny edge detection algorithm [2], that simplifies the next steps. The threshold parameters that are to be found empirically are set to be 30 and 70, which proved to give good results on various images.

[1] The FEN in the file is not correct as the example does not intend to show the piece identifier precision, but the format itself.

Example:





Page image *Canny featured image*

6.2.2 Retrieving contours on the image

The next step is finding contours on the image [3]. Only extreme outer contours were extracted and only end points of them are stored in order to save memory.

6.2.3 Getting rectangles

For each contour, bounding rectangle is found. The initial criteria against the rectangle being a diagram:

1. The relative width and height of the rectangle is higher than the given number;
2. The absolute difference between rectangle's height and width is higher than the given number.

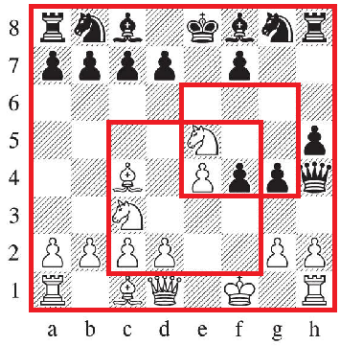
Rectangles that do not satisfy these criteria are stored for the future analysis.

The maximum absolute difference between rectangle's height and width is 5 pixels.

The lower bound of rectangle relative size is $\frac{1}{7}$.

6.2.4 Ruling out intersections

The algorithm steps described above proved to successfully find diagrams positions. However, sometimes the problem emerged:



7...♘h6

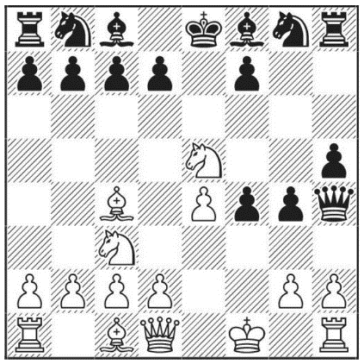
Or 7...♖h7 8.♘d5 ♕d8 9.d4 d6 10.♘d3↑ What's the assessment? 'Unclear' or 'Initiative to White'? The answer is 'both' but if you don't prefer White then 2.f4 is not the move for you.

Problem example

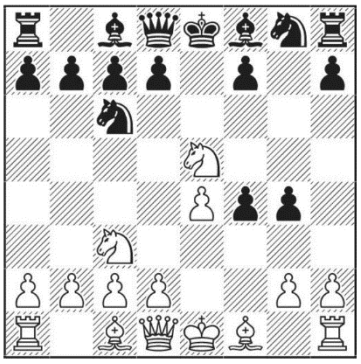
The solution to this problem is finding intersections between found rectangles. If two rectangles intersect, only one with the largest area is stored.

6.2.5 Example

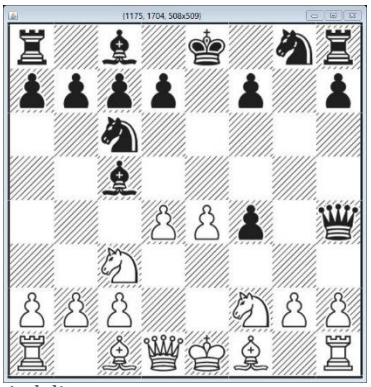
Diagrams shown in this section have been detected on the image example above.



1st diagram



2nd diagram



1rd diagram

2th diagram

6.3 CELLS DETECTION

As it was stated before, diagram detection does not work perfectly in all cases. Sometimes it names other square images as diagrams; sometimes it cuts diagrams with extra borders. In order to overcome this difficulty, an algorithm was devised.

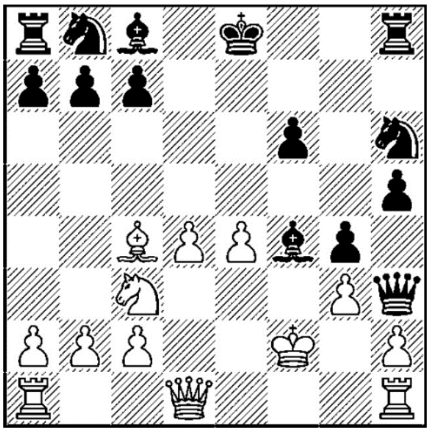
6.3.1 Detecting edges

Diagram images extracted on the previous step are stored in grayscale format. However, it is not convenient to analyze image that way, thus the first step of diagram processing is binarization [4].

At this point, no adaptive methods have been applied. The threshold for the binarization is 170.

The next step is similar to binarization. It is a Canny algorithm [2]. After image binarization, it is possible to set algorithm's threshold higher, yet still find all the edges. The thresholds are 80 and 120.

Example:



Binarized image

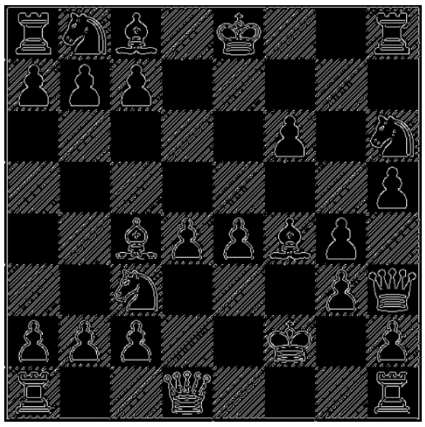


Image after applying Canny algorithm

6.3.2 Finding lines

The idea of the algorithm is in the fact that while scanning image left to right and up to bottom we sometimes come across number of consecutively black pixels. So the next step is to find those "lines" and store them.

The algorithm is pretty brutal: loop over image rows and columns and store lines in two arrays. For each line its starting point and length is stored.

However, we do not want too short or too long lines to be saved. Thus for each line we check if its length is close to the approximate length of the cell.

The approximate length of the cell is $0.1 * \frac{\text{diag\_cols}}{8}$. This value is quite large and ensures that each line inside a square fits the criterion.

6.3.3 Getting cuts

Here "cut" is a line that separates rows or columns. The aim of the algorithm is to find them. If the number of row and column cuts are exactly nine (the right-most and left-most edges are detected this way as well), then the image is truly a diagram and cell images can be extracted from there by the cuts.

The method of finding cuts remains the same[2] for rows and columns and includes the following steps:

1. Sorting lines by the x coordinate of the starting point;
2. For each coordinate that is a starting point all lines starting from there are stored, sorted by their length;
3. For each set of lines the longest sequence of numbers close to the approximate length of the cell is found;
4. If found sequence is longer than the given number, it is considered to be a cut, so its x coordinate and length are stored;
5. If the cut's x coordinate is close to the previous cut's length + x_coordinate value, both cuts are added to the answer;
6. Finally, it is verified that the number of cuts equals nine for both rows and columns.

6.3.4 Extracting cells

After the previous step, cell extraction is pretty straightforward. The image is cut according to the found cuts and cell colors are found as described below.

If n equals 0, then the color of the cell is black, otherwise it is white, where n is found by the formula

$$n = (\text{row\_number} + \text{col\_number}) \bmod 2$$

Row_number and col_number are number of row and column from 0 to 7 respectively.

6.4 CELL PURIFICATION

That is considered to be the most important part of the program. The purpose of these algorithms is to clear cells and transform them to the standard form. The standardization process is described in this chapter.

6.4.1 Binarization

In this part of the program, binarization is crucial. It is not enough to set threshold to some estimated value. Using adaptive binarization methods [5] [6] do not give desirable results as well.

[2] Column cuts are found exactly as described; in order to find row cuts the diagram is simply rotated.

It was empirically discovered, that the color of piece is darker than the color of noise, hatching and background. With this knowledge, the following algorithm was created:

1. Getting average image color and storing it as *upper_threshold*;
2. Finding pixels darker than the average color multiplied by a coefficient that depends on the cell color;
3. Sorting all dark pixels;
4. If the difference between consecutive pixels stored in the sorted list is greater than one, the color of the darker pixel is stored as *lower_threshold*;
5. Looping from *lower_threshold* to *upper_threshold* and applying binarization algorithm, it is possible to find the best *threshold* by counting white and black cells. Once their absolute difference is lower than the given number, the *threshold* is found.

The "given number" in the algorithm is calculated by formula

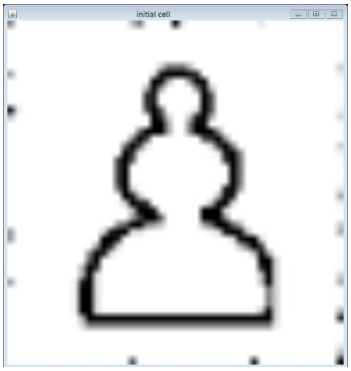
$$\text{img\_columns} * \text{img\_rows} * \frac{9}{10}$$

and was found empirically.

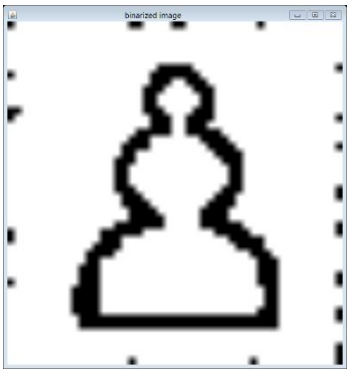
6.4.2 Flood filling

The next step is flood filling the image. In order to do this connectivity components are found on the image. If it relative area is lower than the given number, the area is flooded with the color of its border.

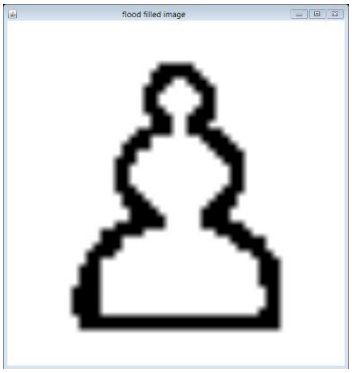
6.4.3 Examples



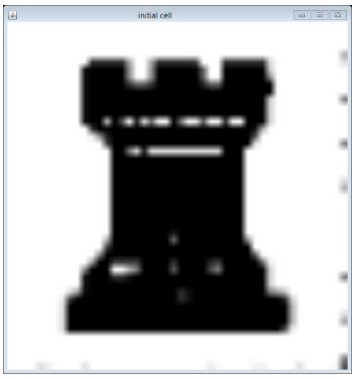
1. Initial image



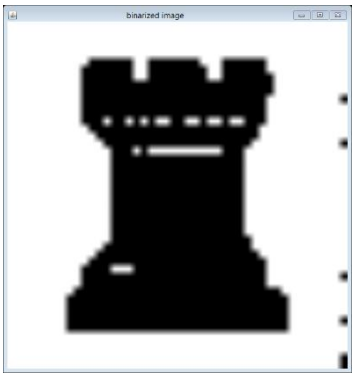
1. Binarized image



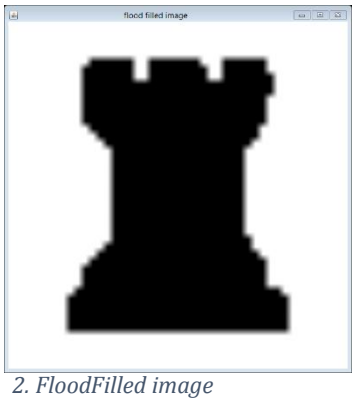
1. FloodFilled image



2. Initial image



2. Binarized image



2. FloodFilled image

3. Initial image

3. Binarized image

3. FloodFilled image

4. Initial image

4. Binarized image

3. FloodFilled image

6.5 PIECE COLOR RECOGNITION

There are two methods to decide whether piece color is white or not.

1. Average color of binarized image;
2. White area floodFilled inside a piece.

If average color of the image is low enough or size of the floodFiiled area is big enough, the piece is considered to be white. Otherwise it is black.

6.6 PREPARING DATASET

Cell examples are detected by automatically extracting cell images and manually describing them. For each cell .png file there is .txt file with information about cell color, piece color and piece value.

The dataset should also be transformed to the uniform format.

6.7 SIMPLE PIECE RECOGNITION

The easiest way to check if the piece on the cell equals to one from the dataset is pixel by pixel comparison. This method had been brutally tested and gave results above the random selection.

Random selection have the accuracy score = $\frac{1}{6*2+1} * 100\% \approx 8\%$

6.8 MACHINE LEARNING METHODS

Feature extraction and classification teaching will be performed using OpenCV and Weka libraries.

6.9 EMBEDDING INTO THE APPLICATION

The necessary steps to be made:

1. Make sure used libraries are supported by Android;
2. Add used libraries into Maven package;
3. Embed project into the application;
4. Improve algorithms in speed and memory usage.

7 TECHNOLOGY

7.1 PROGRAMMING LANGUAGE AND ENVIRONMENT

The only programming language used is Java, as this language is the easiest to support while creating Android application. There are also several handy libraries that can be added to Java project easy enough. Java version 8 has been used.

The GUI for the coding is Intellij Idea, ver.14.1. It proved to be maybe not fast, but stable enough, with many useful features, such as:

- Easy maven integration;
- Git synchronization;

- UML diagram constructions;
- TODO syntax.

7.2 PLUGINS AND LIBRARIES

Fortunately, there are a lot of libraries and plugins that eliminate the need to implement some of the well-known algorithms or setup project manually. The most used in this project are:

- Maven – project compiler;
- OpenCV – image recognition library;
- ghost4j – PDf parser.

8 DIFFICULTIES

During the work on the research several problems were encountered.

8.1 OPENCV LIBRARY

OpenCV is a great library that provides handy API and a lot of implemented algorithms. However, although C++ and Python documentation is full enough, documentation on Java API is poor and mostly consists of code examples on C++. It was often not obvious how to rewrite the code on Java due to Java features, for example operator overloading is not supported.

8.2 SPEED AND MEMORY

Java is infamous for its low speed and memory usage. In some cases, there was not enough memory so it was necessary to split work in stages, which slowed down the project.

8.3 HATCHING

In many books, black cells are depicted as white with hatch.

Ex. 1

Ex. 2

Ex. 3

Before the idea of floodFilling came to mind, there were attempts to remove lines using various methods, such as Hough line transform [7].

8.4 DIAGRAM VARIETY

As there is no standard way of depicting diagrams, the most difficult task is to make the algorithms work on every possible image. That is why it is not enough to simply use any classification algorithm, but additional methods have to be applied.

8.5 PREPARING DATASET

Although dataset images were extracted automatically, their descriptions had to be made manually. The task is tedious and time-consuming. There are possibly some mistakes in there, as after some time data was not extracted very carefully but only as fast as possible to finish the task already.

8.6 EMPIRICISM

Although the algorithms described above work good on the books chosen to be in the dataset, it is possible that in some cases there will be issues, as many parameters were set based on personal observations empirically.

9 RESULTS

Despite the difficulties that had to be overcome, during this term it was possible to create a complete Java project supporting the following features:

1. Transforming PDF books into specific format;
2. Diagram detection on the image;
3. Cell detection algorithms;
4. Cell purification*;
5. Prepared dataset*;
6. Simple piece recognition.

The features marked by "*" still have room to improvement.

10 FURTHER RESEARCH

Although many necessary features have been implemented, there is still a long way ahead. The work on this project will continue in future. The current TODO list consists of the following:

1. Improved algorithms marks by "*" in the previous chapter;
2. DJVU support;
3. Android integration;
4. Casting dataset to the uniform format;
5. Creating good classifier and improving piece recognition.

11 RELATED WORK

As it was mentioned before, there are several works on chess recognition and applications for chess book reading. However, they do not fully support all features authors' project is going to provide.

The example of an application for book reading can be found on Google Play and is called "Chess book study" [8]. Although it positions itself as a chess book reader, the only supported feature is an interactive diagram that takes 1/3 of the screen, cannot be removed and pieces on which user has to place manually, which is really long process.

One of the chess recognition process is Chess Grabber by Dion Nocolaas [9]. This project support diagram recognition and creates FEN from chess diagram picture. However, the algorithm implemented by author are not perfect, as he admits himself. What is more, the website recognizes only diagram images and to not provide diagram detection and book parsing.

Thanks to robust image recognitions developments, a lot of algorithms have been devised that solve various intermediate problems that come along with the task. Information about the papers read while working through project development can be found in the reference section below.

12 CONCLUSION

Although chess recognition can be a difficult task due to the variety of diagram types, it is still possible to develop an algorithm that recognize chess pieces and chess diagrams on the images. However, if the project is to be integrated into Android application, it should not be memory- and time-consuming.

13 LINKS

The source code of the project can be found on https://github.com/cscenter/interactive-chess-viewer-2.

The source code of the image recognition part is on https://github.com/cscenter/interactive-chess-viewer-2/tree/master/Image%20recognition.

Slides of the presentation and screencast can be found on https://github.com/cscenter/interactive-chess-viewer-2/tree/master/Image%20recognition.

14 REFERENCES

[1] Kirill Kryukov, "Forsyth-Edwards Notation," [Online]. Available: http://kirill-kryukov.com/chess/doc/fen.html.

[2] JOHN CANNY, "A Computational Approach to Edge Detection," in *TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 1986.

[3] S. a. A. K. Suzuki, "Topological Structural Analysis of Digitized Binary Images by Border Following," pp. 32-46, 1985.

[4] asif, "Conversion of a Color Image to a Binary Image," 4 April 2005. [Online]. Available: http://web.archive.org/web/20080610170124/http://www.codersource.net/csharp_color_image_to_binary.aspx.

[5] OpenCV development team, "OpenCV documentation," 25 Feb 2015. [Online]. Available: http://docs.opencv.org/modules/imgproc/doc/miscellaneous_transformations.html.

[6] J. Sauvola, "Adaptive document binarization," *Document Analysis and Recognition,* pp. 147-152 vol.1, 18-20 August 1997.

[7] J. a. G. C. a. K. J. Matas, "Robust Detection of Lines Using the Progressive Probabilistic Hough Transform," pp. 119-137, 2000.

[8] A. Pereira, *Chess Book Study,* Kadugodi Bangalore, 2014.

[9] D. Nicolaas, "Chess Grabber," Dion Nicolaas, 2012. [Online]. Available: http://chessgrabber.nicolaas.net/.