Tracking and annotating a chess game

Georgi Petkov

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2013

Abstract

The goal of This honours project is to develop algorithms to track and annotate a chess game using images taken from a real camera, as well as to provide extensions such as Graphical User Interface, Simulator and other features that can be used to further extend the functionality of this application. As a more specific sub-goal the project is focused in investigating how successfully a model-based logical algorithm can be designed to capture the whole variety of chess combinations under the given circumstances. The algorithm achieved 99,01% accuracy over 54 games and 3680 moves analysed in total.

Table of Contents

Chapter 1

Introduction

The main goal of the project is to annotate a chess game using a camera observing the chessboard. Both the chessboard and the camera are located in the Informatics Forum building which presents some challenges such as changing lighting, accounting for human players moving the pieces, shadows and the possibility of having pieces partially or completely hidden behind other pieces due to the angle of observation and the difference in their sizes. All these constraints require this project to be treated as something more than a visual processing application. Due to frequently unclear visual input from the camera it is very unlikely that approach based on visual processing mainly would turn to be successful. Striving for stability and best possible accuracy, I decided to build the core algorithms around logical model-based mechanism, extensively using the rules of chess, and apply vision processing techniques to additionally support this model.
The formal requirements of the project are to deliver algorithms that are addressing the following problems:

- Detection of a moved piece (or pieces during a capture).
- Identification of a moved piece.
- Transcription of the sequence of the movements.

My ideas and understanding of the problem led to the derivation of a number of stages that need to be completed in order to capture the requirements well and provide any additional functionality that was necessary.

1.1 Detection of a moved piece.

The first part of the algorithm should be able to get some suitable snapshots from the camera and use them to produce an annotation of the moves. The application I have developed is able to get screenshots manually by pressing a button or activate an automatic screen capture which takes screenshots every few seconds. The images are then processed for threshold adjustment and the clear images are forwarded to the

move detection algorithms for further analysis. The screen capture is implemented by invoking a script that gets a real-time snapshot from the camera and stores it in a folder specified by the application.

1.2 Identifying regions of change and finding moved pieces.

The first task of the algorithm related to finding moved pieces is to compare the old and the current images pixel by pixel for changes in their RGB color values and identify the potential regions of change created by moving a piece on the board. The difficulty comes from selecting two snapshots that provide meaningful information and disregard the presence of human beings or other noise factors on the board. The real challenge was to create a stable and accurate algorithm that is able to reason about what move had been made and then translate that reasoning to any standard chess notation. My algorithm is model-based oriented rather than visual processing oriented and attempts to use the logic of chess in its advantage. Having the heavy work of identifying moves done in this fashion has the advantage of being as independent as possible from changing environmental factors that would normally cause the visual approach to fail in many cases.

A crucial part of the algorithm is to identify what piece has actually moved from one square to another. In addition to any computer vision techniques the basic rules of chess can also be used to disambiguate moves by checking them for consistency. This works only by making the assumption that every move in the game is a valid one by the official rules of the game. Identifying which piece has actually moved is quite often a very difficult task that requires a lot of reasoning and consistency checks and can require many branches of the algorithm to provide specific checks and at the same time work in every general case. As this is one of the most important parts of the project many different approaches had to be discussed, attempted and evaluated. I tried to investigate what are the strengths and weakness of every single one of them and see if they can be used in combination to obtain optimal performance for every general situation. More detailed presentation of all considered approaches are discussed in Chapter System overview - Comparison of different methods.

As the moving detection algorithm is one of the core elements in my project detailed explanations about its specific functions will be provided at the Moving detection chapter.

1.3 Record keeping and consistency checking.

The high possibility of moves that cant be disambiguated immediately requires more than one record of the game progress to be stored for further consistency checks. The intuitive idea is to store the game progress as a tree structure branching every time there

are more than one possible annotations. After some tests on different implementation of such data structure it turned to be sufficient to keep only a candidate pair of solutions containing two squares - where the piece has moved from and where the piece has moved to, including some other fields and a reference to a child object of the same type. The idea behind consistency checking is to keep the tree with recorded moves neat and clean in a sense that it should eliminate moves proven to be inconsistent with the following ones and eventually provides only one valid annotation. It is important to note that although keeping alternative solutions can sometimes be the only right option, my approach is to avoid this as much as possible as it often leads to undecidable situations until the very end of the game. The problem comes from the fact that tolerating branches generally leads to even more branches - an algorithm that cannot identify a single most probable solution for a certain move has no guarantee to do so at later stages. Although this claim can be supported with theoretical reasoning my main reason to include it here is that it was also very strongly confirmed by empirical data and testing.

1.4 Game annotation.

After researching different options I chose to use the standard PGN notation for annotating the games analysed by the algorithms. The difficulties in the translation come from accounting for all possible situations such as castles, en passant pawn takes, pawn promotion, check, checkmate, etc. It is not at all a trivial task to design algorithm that can accurately translate the data structures into notation and in some cases even with correct work of the move detection algorithms the annotation can still fail at the testing stage. That is why in discussing the project in details the translation algorithm has to be explained in at least the same level of detail as the moving detection algorithm. It is important to note that as move detection, providing a transcript of the game is a core functional requirement of the project.

1.5 Board representation.

Although it is not present in the functional requirements of the project I considered that it is important part of succeeding in the above tasks to have a clear idea of the board representation throughout the design and implementation of the project. That also motivated my choice of Java due to the convenience it offers in designing intuitive and practical GUI components.

Speaking about board representation related to this project there are two different concepts that need attention and disambiguation. The board is generally represented in two different ways in a running application:

- As an image on the program interface.
- As a data structure in the program memory.

Both of them are equally important and do not always correspond completely to the same game records. The relations between them are simple if explained in detail but can be confusing if no differentiation is made between work in Real data and work in Simulator mode. I have not yet introduced the idea of a simulator but it needs to be said that while in Real Data mode the image of the board is the only true data representation and the data structure interpretation of it might contain errors due to wrong translation, in simulator mode there it is almost certain that the image of the board corresponds completely to the board data structure held in the program memory.

1.6 The chess simulator

An invaluable tool for testing, validating and improving the performance of the program, the chess simulator is another extension I decided to bring into the project. It provides great assistance not only to designing the algorithm and capturing rare cases but also for testing and validating the whole project performance. Due to the fact I am implementing the model-based approach completely on my own I had to design the simulator on my own as well which took significant amount of time but the efforts were quickly paid off by the new functionality it offered for quick testing and identifying problems with the algorithm. The main need of a simulator came from the fact that even if I provided some sort of algorithm claiming that it solves the problem, testing it on real data would be time consuming, insufficient in case that only a few games are played and thus inadequate to prove that my algorithm is stable and applicable in general. The same problem holds not only for the final testing but for the design and implementation of the algorithm as well. The simulator is one of the first modules I built for this project having the idea of test-driven development and it turned to be invaluable tool for evaluating and improving the performance of the move detection and translation algorithms. Due to its huge importance for the project a special chapter is dedicated to explaining how the simulator works in details.

1.7 Research and background reading.

1. B. Majecka, ”Statistical models of pedestrian behaviour in the Forum”, MSc Dissertation, School of Informatics, University of Edinburgh, 2009.

   This paper was on a similar problem - dealing with visual data in real environment where changing light levels, shadows, obscurations are a constant problem. B. Majecka had a very successful method of background equilibration that seem to have dealt quite well with the problems above. A version of this method may become really useful in my application if it turns out that the tests with real data in different times of the day produces very significant differences in the accuracy of the algorithm .

2. Introduction to Vision and Robotics lecture notes.

I had to do some reading from the IVR notes in order to have an initial understanding of computer vision, background subtraction, homography transformations and object recognition. These are more specifically lectures 2-5. At the beginning one of my first attempts at the project was implementing a homography transformation and seeing whether that can be useful for improving the accuracy of the algorithm. At this time the statistical data is not enough to confirm whether it is better to use the homography image or the original one.

3. Online materials on official chess notation.

   I had to familiarise myself with different chess notations and make a decision about what it is best to use for my project. In a way, totally symbolic (without any letters) notation systems could be better as they work for any language but the popularity of the PGN annotation system and the idea I had later on for importing and exporting PGN files was exactly what was needed for the project.

   Example of PGN notation follows:

   1. e4 Nf6 2. e5 Ng8 3. Nf3 Nc6 4. Nc3 Nb8 5. d4 Nc6 6. Bb5 Nb8 7. O-O a6 8. Ba4 b5 9. Bb3 Bb7 10. Ng5 e6 11. d5 h6 12. Nxf7 Kxf7 13. Qh5+ g6 14. Qf3+ Kg7 15. Qh3 exd5 16. Bxh6+ Nxh6 17. Nxd5 Bxd5 18. Bxd5 c6 19. Be4 Nf7 20. Qg4 Nxe5 21. Qg5 Qxg5 22. f4 0-1

   Most of my research here comes from chess websites such as www.gameknot.com and Wikipedia.

4. Homography transformation.

   I had to do some reading on homography transformations separate from the material in the IVR lecture notes. Although, it is very well presented there I wanted to see more examples of applying homography and check some forums for java implementation of it. Eventually Ive made a java implementation of the homography algorithm that seems to produce the expected results.

5. JAVA GUI layouts and components.

   From the very beginning I am working on improving the layout and GUI design of the application and I am constantly checking for ways to improve the functionality of my interface and make it easy to use. This requires not only extra reading on the description of the java components but also actively looking for examples of chess-like applications for more ideas on how a neat and efficient layout looks like.

6. Examples and discussions on coding allowed moves for different chess pieces.

   This is a very tricky problem as the complexity of the piece interactions in a game of chess can actually be very high. What is essential for the work of the program is to find a way to code the basic allowed moves of each piece so it is easier to predict what piece could have possibly moved from one square to another even without convincing image analysis proves that this had actually happened. Such methods play a very important role in resolving inconsistencies as well as their direct application would show whether the current move makes

any sense at all given the previous ones. Many different chess applications that I was able to check use their own complicated systems of coding the allowed moves of the pieces but fortunately for the goals of the project even a simplified version is sufficient.

Chapter 2

System overview.

2.1 Strategic decisions.

Choice of Programming Language: Analysing the complexity of the project goals and forming an idea about the final delivery of the algorithms were important considerations to take into account when choosing the language for implementing the solution. Although, C and Matlab especially can be very powerful in working with image data my primary choice was Java. The reason for this is that Java can efficiently be used for solving any computer vision problems, has strong Object-oriented behaviour that can be applied in designing the model-based move detection, data structures and it also provides tools and libraries for project management and GUI design.

Choice of official chess notation: Standard Portable Game Notation (PGN). PGN is a plain text computer-processable format for recording chess games (both the moves and related data), supported by many chess programs.

Choice of platform: Eclipse. My goal was to deliver a stable and extensible application open to adding extra functionality. The complexity of the problem on its own and my test-driven programming approach needed a good platform like Eclipse to keep the version control and debugging issues easily manageable. As the development of such algorithm requires endless testing and re-factoring the code my debugging strategy had to be efficient. I made use of the option to output the console to a file using informative print line statements on key execution points of the algorithm. This approach, combined with the debugging tools of Eclipse, turned to be very successful for having solid and continuous progress.

2.2 Discussion about different methods and justification of the selected ones.

I would like to provide a brief overview of some of the strong candidates for approaches for solving the move detection problem:

- Use image analysis and pattern recognition to identify what piece have moved and where: This approach seems very intuitive and useful in the beginning but it turned out that it is unable to deliver high accuracy and be the main algorithm to rely on. There are many reasons for that but the most important are listed below:
  - Noise in the image - this a permanent issue when taking screenshots from the camera due to: changing lighting; shadows cast on the field by humans, furniture around or the pieces themselves; rainy or foggy weather (the chessboard is outside). It is very hard to account for all conditions that may add noise to the image and if the algorithm depends mainly or image processing the accuracy might be unpredictable and being totally out of control as it heavily depends on external factors.
  - Bigger pieces hiding completely smaller pieces, which is a solid reason why standard image recognition techniques would not work for finding which piece has moved. Sometimes the area of interest is just hidden behind others and can not be forwarded for any useful pattern recognition.
- Implement and train an AI algorithm to analyse the moves and use prediction probabilities to guess what move has been made based on the incoming image data. This approach has many advantages and seemed like a very strong candidate that could potentially work in every general case. Its implementation would require the use of a chess engine not only to help with calculating the probabilities but also to generate computer vs computer games for training the AI algorithm. Unfortunately, it may have significant downsides which forced me to look for alternative solutions.

  The AI algorithm does not provide enough control on specific delicate moves that are common in every chess game such as: En Passant pawn take, pawn promotion, possibly would face problems with castle situations. As most of the training data would consist of general moves unordinary moves would have very low probability of being considered which would provide significant disadvantages of the algorithm specifically if wrong suggestion such as En Passant pawn take is made at the very beginning of the game. I decided that such uncontrollable behaviour should be avoided and replaced with simple and logical reasoning approach providing attention to the very low details in the chess game. The AI algorithm might be hard to train and easily over-trained. It would be trained by computers as this is the only way to generate big enough set of training data and it may provide wrong

assumptions about the probability of making certain move as humans generally think more strategically and intuitively than computers. Non-logical moves such as piece sacrifices would have very low probabilities and would hardly get picked over more logical moves which the AI would generally favour. The last disadvantage is that using such a method would force me to use extensively chess engines that are not designed by me which would imply some external constraints. It also leads to reduced freedom and creativity in the implementation of the algorithm which is a situation I decided to avoid.

The optimal strategy seems to be doing as much analysis as possible to reach only one conclusion that is most probably and forward it for translation and add alternative solutions to a tree structure recording the game only where severe uncertainty occurs. In that case the game can be possibly verified at later stage and after making a valid consistency check wrong suggestions can be eliminated. This is a reasoning model-based approach, using detailed analysis of general moves and special cases to produce solid logical algorithm that indeed proves to bring the accuracy to a very high level.

There was an idea to use extensively tree data structures and perform consistency checks at later stages using the assumption that further moves would validate certain branches in favour of others but the implementation of this method proved that assumption wrong for the most part. Especially at later games there are not enough constraints due to less active pieces on the board to select the correct branch with high probability.

2.3 Abstract model view of the chessboard vs visual data.

Both the model based view of the chessboard and the image analysis on the visual data from the camera are crucial to the correct work of the annotation algorithm in such environment. As two algorithms based on each of these two methods can have totally different behaviour it is necessary to prioritise one of the methods above the other. The strength of the model-based approach comes from having information about the initial setup of the pieces in the game and how they are allowed to move. This provides a very strong platform for reasoning and further assisted by visual analysis where necessary has the potential of delivering very accurate transcript of the game avoiding most of the invariables related to lights, shadows, etc that are a big area of potential problems for the image analysis.

2.4 Graphical User Interface - The Digital Chess Observer.

One of the good features in this project is the user interface which provides almost full control over the application. It serves several different purposes that helped me in the implementation and debugging stages of designing the algorithm. Although its main function was initially to serve as an assisting tool for programming, the GUI has many other features that are easy enough to use and understand even after the development has been completed. It plays more than one role in this application and I would like to give some insight in some of them:

2.4.1 Game observer.

As implied by its name one of the primary role of the GUI is to be an observer of the chess game. Both in Real-data and Simulator mode the three image panels would constantly update their images to correspond to the current moves in the game. Every image panel provides a little different perspective into the game:

- Main panel:
  - Simulator mode - shows the current picture generated from the PGN notation.
  - Real data mode - shows the screenshot taken from the camera at the Informatics Forum.
- Move detection panel - highlights the areas of change between the last two pictures.
- Homography panel - shows a homography transformation of the image at the main panel so it help seeing what move exactly has been made.

2.4.2 Printing the game transcription on the screen.

The GUI automatically provides the game annotation in a list ordered by move number on every turn. Although the annotation is also printed in the analysis files, by printing the annotation on the screen the GUI fulfils one of the most important requirements of the project. This is also very useful as it allows the annotation to be checked against the actual image on the board for correctness.

2.4.3 Providing control over the application.

Several buttons and menus provide control over the application. They allow the following important actions:

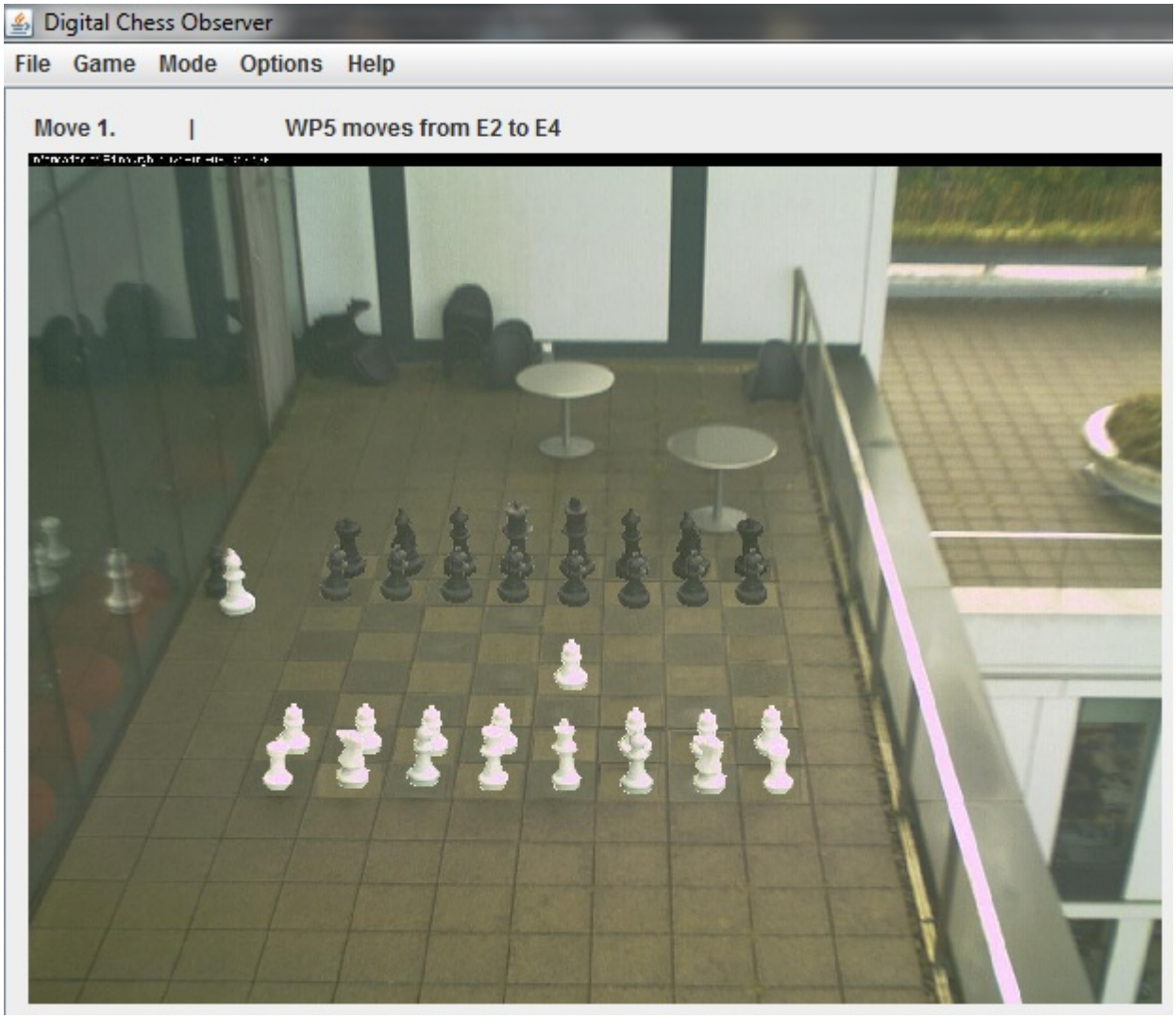


Figure 1. GUI - Image panel.

Figure 2. Move detection panel.

- Switching between Simulator and Data mode. Simulator mode is the default one as it needs no preparation and can start analysis immediately while data mode is not always possible - it requires access to the camera and somebody actually playing a game.
- Starting a test analysis on a data set of PGN games. This is initiated by the button Play Simulator and requires an imported record of PGN games.
- Importing PGN games.
- Advance the game one step in Simulator or Data mode. Very useful for following the game and checking the accuracy of the transcription manually.
- Take screenshot now.
- Start Real data automatic annotation.

2.4.4 Setting up important program parameters.

The interface provides options for setting the program variables and important data for initialising the board data structure and the algorithms precise behaviour. Although, I may decide to hide this functionality from the final version it serves a good role in making the application extensible and applicable to other such programs located at different places than the Informatics Forum. They would require a little bit different setup and identifying this from early stage and making it possible to change these settings is a desirable feature of every application as it plans for change and future extensibility.

2.5 Program variables:

The specific details of the program variables are not so important at this stage but it is good to have a rough idea of them to understand the underlying mechanisms of the algorithms. Some of the most important program variables are explained below:

(a) **Threshold for the distance around a point** (typically a centre of a board square) - Defined in *class PointOld* - method clicked() - threshold for the distance around the point currently set to *7 pixels*. This variable is very important as it influences the choice of candidate squares affected by changes in pixel values at the beginning of the move detection algorithm. Increasing its value would lead to more candidates to be allowed for analysis while decreasing it may omit possible correct solutions. The value of 7 px is experimentally chosen after evaluating more than 3680 moves on the simulator and scoring best results with that value. Changing this variable may require changes in the code as well!

(b) **Colour value threshold for Move detection** - this variable sets the range around pixel RGB colour change when comparing two images while tracking chess moves. Such variable is required only when working with real data due to noise in the image that is not present in Simulator mode. Dilate and erode functions are typically quite useful for removing noisy pixels from the image but it turns out that a precise value for the threshold variable in this case is sufficient to reduce the noise significantly. This is due to the fact that change in the colour of the pixels triggered by an actual move is a lot more significant than changing light or other noise factors.

(c) **Field end points** - they specify the endpoints of the chess field and in case the application is used on a different setting they need to be set so they correspond to the new environment. They play an important role in providing nice homography transformation of the board typically shown on the third image panel, that attempts to provide better view on the otherwise tilted board.

(d) **Square centres** - an array of 64 point objects holding the location of every centre of a square. These variables are very important for the work of the algorithm and always have to be initialised with care. The GUI provides functionality for easily selecting the points and storing them in a file in case they need to be changed or adjusted slightly differently for some purpose. Example for motivation of such intentions would be displacing them slightly to add noise for the simulator and test its work in harder conditions or simply test the algorithm in completely new environment.

2.6 Motivation for building a simulator.

Even before making any significant progress in designing the algorithm it was clear that the testing and validation strategy needs to be established carefully. The problem came from the fact that it is insufficient for proving that the algorithm works satisfactory if the set of testing data is too small. A summary of why is hard to design such algorithm using real data is listed below:

- Moving the pieces, taking good screenshots and evaluating the performance of the algorithm takes a lot of time and requires additional human resources.
- Completing even one game on the actual board is very slow and does not guarantee enough variation.
- Building a good algorithm requires constant improvement and re-evaluating it on the same set of data. Such repetition again requires too much resources as at least one other person is required to assist with the movement of pieces.

Therefore I chose to create a simulator that can import chess games recorded in PGN notation, draw an artificial board looking as close as possible to the real

one and use it to evaluate the algorithm. This was very beneficial in terms of improving the performance of the algorithm using wide variety of games and eventually have some statistics and results to analyse.

2.7 Program structure.

The whole chess project is split into three main packages:

- chess.board - the most important package for the project. Contains all the functionality needed to work on Simulator and Real-data mode as well as the MoveDetection and Annotation algorithms.
  - class Board
  - class CandidateTple
  - class MoveDetector
  - class Piece
  - class Sqare
  - class RealData
  - class Simulator
  - class ScreenCaptre
- chess.gui - contains the Main class and all components for the GUI.
  - class Main
  - class ImagePanel
  - class ToolbarPanel
  - class StatusBar
  - class FileMenu
- chess.utils - a package containing support classes such as Tools, MouseHandler, FileManager, PointOld, Homography, etc.

The Class diagram illustrated in this section shows the relations between some of the main classes in the program. The Simulator and Real-Data class are independent and both work with the MoveDetector class for analysing moves and annotating them. The ImagePanel class is accessible from both the Simulator and the Real-Data class and the control over it is shared by setting the Boolean variable isSimulator. The same variable controls the access to the MoveDetector class. The diagram also shows that at any given point of execution of the program all four classes are sharing the same instance of the Board class. This is designed on purpose for future extensibility of the Simulator functionality into Real-Data mode aiming at improving the accuracy of the predictions.

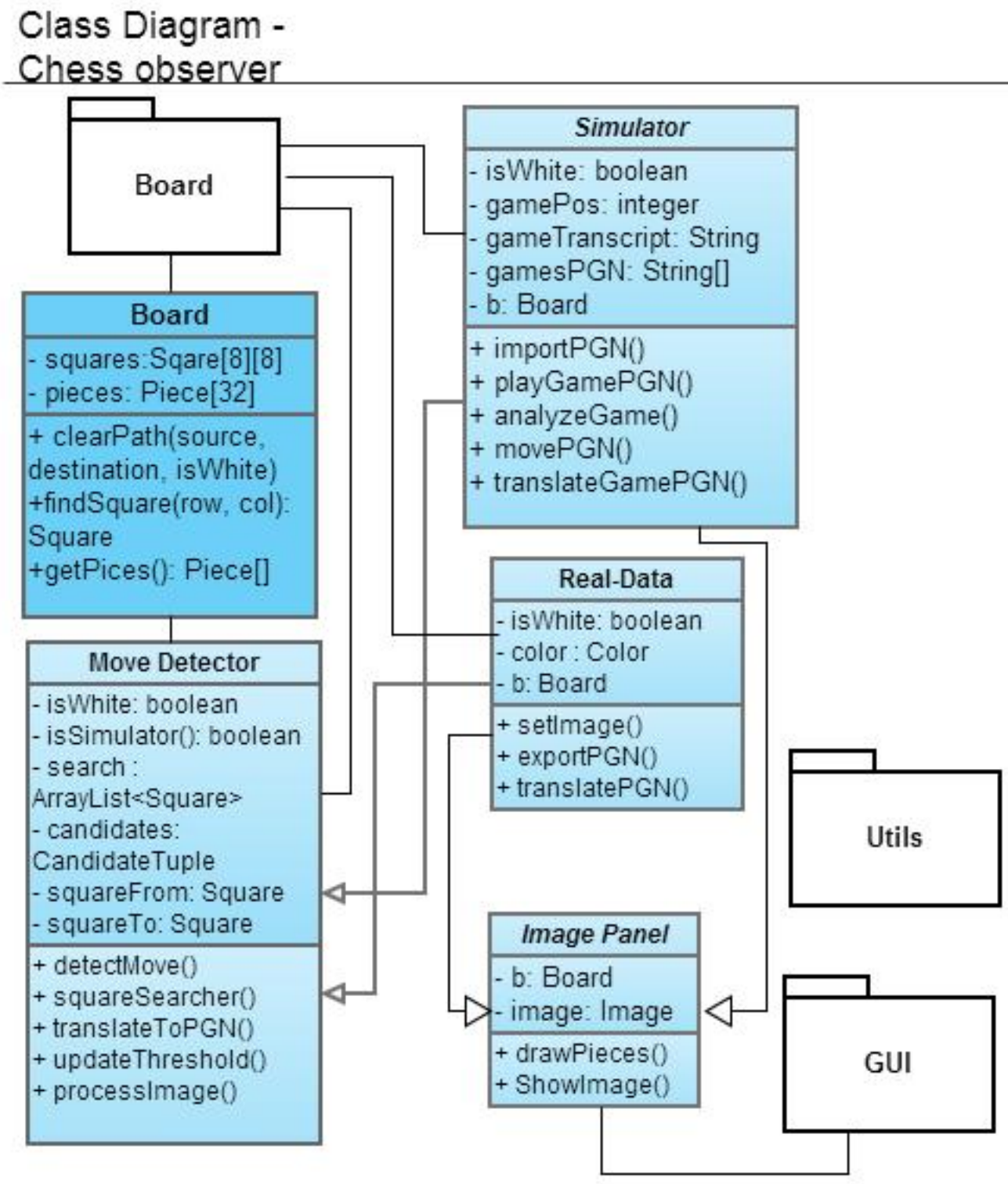


Figure 3. Class diagram.

2.7.1 Important program directories and folders

If the program is started from an executable jar file, some folders and files have to be prepared in order to work properly. That is why I would like to mention them explicitly in a few sentences. Assuming that the the executable is in folder called chess, here is a list of folders and files that must be present there as well.

- Folders:
  - file: must contain MID_POINTS file. This file is used to initialise the square centres and its presence is crucial to the work of the algorithm.
  - pgn: contains the PGN records that the Simulator imports for evaluation of the algorithms.
  - analysis: this is where the Simulator stores the test results.
  - images: must containg EMPTY_BOARD.jpg and the prefered set of chess pieces for the Simulator.
  - newCaptures - currently holds the images for the tests on real data with changingLighting.

2.7.2 Design and implementation.

This chapter is focused on the design of the whole application including data structures, algorithms and GUI. Although, most of the code is already implemented and for the moment works satisfactory there are parts that are constantly being improved and they are relevant only to the current stage of the project.

2.8 Data structures.

2.8.1 Game records Data Structure.

This is a tree data structure that connects each parent node, representing the previous move, with one or more nodes which are the potential correct annotations of the game. Each node can either hold the PGN notation for a given move or another object containing more data. It is hard to predict from now how the end version of this data structure would look like as changes in the rest of the code affect its complexity and level of detail.

2.8.2 Chessboard Data structure.

The Chessboard data structure is a key part of the model-based method for detecting moved pieces. There are two intuitive approaches for holding relevant

information about the state of the board:

2.8.2.1 Square object:

The Chessboard object contains an 8x8 Square array. Every Square object records the following data:

- Piece on this square - indicated by a piece code from the following groups:
- Black piece codes: BR1”, ”BN1”, ”BBB”, ”BK”, ”BQ”, ”BWB”, ”BN2”, ”BR2”, ”BP1”, ”BP2”, ”BP3”, ”BP4”, ”BP5”, ”BP6”, ”BP7”, ”BP8”.
- White piece codes: ”WR1”, ”WN1”, ”WBB”, ”WQ”, ”WK”, ”WWB”, ”WN2”, ”WR2”, ”WP1”, ”WP2”, ”WP3”, ”WP4”, ”WP5”, ”WP6”, ”WP7”, ”WP8”.
- Name - The name of the square relevant to the board for example: A1, B2, H8.
- Color - indicates whether it is a black or white square.
- Center - records the coordinates of the center or each square as measured from the camera view.
- Row/Column - assuming that A1 is 1st row, 1st column this variable is used to easily search the array of squares in many algorithms.

2.8.2.2 Piece object

The Chessboard object holds an array of 32 piece objects each of which carries the following data:

- Name: Piece code from the list above.
- Colour.
- Row/Column.
- Captured: Indicates if the piece is still active in the game.
- Promoted: Shows If a pawn has been promoted to another piece. Changes the behaviour of the piece.

Obviously there is some redundancy in using both the Square and Piece objects but my latest changes of the code actually contain both of them and this turns out to be useful for different algorithms. It avoids writing algorithms for searching and makes the abstract board much easier to work with. Unless memory or performance becomes an issue, this strategy will be employed for the rest of the project.

Chapter 3

Simulator

The design and build of the simulator was mainly influenced by the idea of test-driven development of the algorithm. Iterative approach and constant improvement combined with finding specific infrequent cases was possible only by having a huge testing data set. Some of the last tests on the simulator were run over 49 real chess games containing about 5162 moves in total. Testing with the simulator is fast and efficient and has no limitations - any game recorded in PGN can be imported and evaluated with the algorithm.

The testing strategy of the simulator can be explained concisely in a number of separate steps:

(a) Import PGN file containing records of real chess games.

(b) Translate the PGN notation to sequence of commands for moving the pieces on the artificial board - this essentially updates the board data structure and forwards it for drawing.

(c) Draw the pieces according to the information in the data structures and merge the image to produce a picture similar to a screenshot that could be taken from the camera.

(d) Feed the Move Detector algorithm with two such images - one from the previous iteration and one containing the new move.

(e) Run the move detector algorithm and catch errors if any.

(f) Translate the most probable move back into PGN notation and update the board data.

(g) At the end of each game perform test to check how accurately the exported PGN notation matches the original input.

Note that steps 4-6 contain most of the functionality necessary for analysing real chess games using the camera. The testing stage 7 cannot be performed with real data unless a transcript of the moves to be expected is imported in advance. Stages 1 to 3 prepare the simulator for working with the Move Detector

algorithm. The simulator mode was extremely useful in finding special cases thus improving the accuracy of the algorithm. It is also the only way to perform tests on large data set and validate the general correctness of the application.

3.1 Importing PGN.

Extracting PGN records from a file - Importing PGN files is done in the Simulator class. The name of the PGN file has to be specified. The default PGN folder is called pgn and the default filename is record. The File Manager class reads the file and returns a String variable containing all PGN games. The following example shows how a typical PGN record may look like:

[Event ”GameKnot Blitz”]

[Site ”http://gameknot.com/”]

[Date ”2011.03.23”]

[Round ”-”]

[White ”3ff33mm33ss3”]

[Black ”liverpool88”]

[Result ”0-1”]

1. e4 Nf6 2. e5 Ng8 3. Nf3 Nc6 4. Nc3 Nb8 5. d4 Nc6 6. Bb5 Nb8 7. O-O a6 8. Ba4 b5 9. Bb3 Bb7 10. Ng5 e6 11. d5 h6 12. Nxf7 Kxf7 13. Qh5+ g6 14. Qf3+ Kg7 15. Qh3 exd5 16. Bxh6+ Nxh6 17. Nxd5 Bxd5 18. Bxd5 c6 19. Be4 Nf7 20. Qg4 Nxe5 21. Qg5 Qxg5 22. f4 0-1

As there is irrelevant data to the simulator the PGN String needs to be processed in order to store only the games notation in ArrayList so it is ready for automated analysis. The splitting is done first by line and then by space character so eventually every game is represented as an array of String variables containing the atomic Strings of a PGN file - e.g. e3, NxB5, etc.

After the completion of this method the games array contains every single game read from the file.

Extracting a single game record - The method playGamePGN gets a PGN game as a input, which is essentially an element of the games ArrayList, splits it by empty spaces and stores it in new String variable called game. An integer used for iteration through the game is also defined and initialised to 0.

Loading the next move - The main challenge in this method called loadNextMove is to retrieve important information from the PGN string such as:

The move number - e.g. 1, 2, 3, etc. The string containing the actual move data - e.g. e4, Ng3, Qxd5, etc. Determine whether it is Whites or Blacks turn.This is achieved by checking for a move containing dot . character (excluding cases of

e.p - En Passant pawn take), initialising the moveNumber variable with it, then treating the following string as Whites move. Alternatively, moves that do not contain dot are correctly assumed to be blacks.

The correct execution of this method leads to the first step in the simulated board creation - translating the PGN move to a square From and square To tuple.

3.2 Translating PGN into game moves.

Translating PGN games into game notation involves analysing the string and attempting to identify the squares where the pieces might have moved from as well as where they might be attempting to go. Move strings come in a different length - e4, Be6, QxB5, Nexc5+, etc.

That condition requires from the algorithm to account for different lengths of strings and parse them in a unique way. My approach was to split the string into array of characters and then use if-statements to extract the information for each individual character.

The main questions that have to be addressed parsing the move string are the following:

- Which piece is being moved?
- What square it is going to?
- Is it a move to an empty square or piece capture?
- Is it a castle situation?
- Is it a string indicating the end of the game by printing the result?

char[] decode = move.toCharArray();

By analysing the char array decode defined above the information about which piece is being move is easily obtained by taking the first element of the array - decode[0]. This is typically a letter:

- a, b, c, d, e, f, g, h - indicate pawn move.
- N - knight move.
- B - bishop.
- R - rook.
- K - king.
- Q - queen.

The information about the target square is obtained by getting the correct values from the decode array and typically are combination from a letter from a - h and number from 1 - 8.

The array is also checked for special symbols which in the case of PGN notation imply important chess situations.

- x - the piece on the target square is taken.
- + - the move delivers check to the opponent.
- # - the move delivers checkmate to the opponent.
- 0-0 / 0-0-0 - Kingside / Queenside castle.
- e.p - En Passant pawn take.
- Ned7 - the additional letter e in this case specifies that both Knights can move to d7 and the one making the move comes from the E column.
- Ne5d7 - similar to the above situation but in this case both of the nights are positioned on the E column so additional specification is required.

The combinations of the above variations cause the decode array to have lengths varying from 2 to 7 characters. Parsing the data from the array accurately identifies two squares that for simplicity I would call Square-From and Square-To. They refer to the square, where the piece has moved from and the square where the piece is moving to.

In order to proceed to the drawing stage, the list of active pieces has to be revised for candidates corresponding to the required fields. This is done in a method called findPiece() that takes as input a simple code indicating the type of piece which is making the move, the destination square and a Boolean variable indicating whether it is whites to move or blacks to move and returning a piece that can move to that square. Often, more than one piece would fulfil that requirement so the method should be able to remove ambiguity by recursively executing itself until it narrows down the problem to a single most probable solution.

Important parts of this method deal with the following issues:

- Promoted pieces that should also be taking into account and given the opportunity to compete for best candidate. Promoted pieces emerge when a pawn reaches the end of the board and they can be any of the major pieces in the game - Queen, Rook, Bishop or Knight. When a pawn is promoted, the piece is keeping the same ID value but the its name is modified with the special symbol C plus the name of new piece, plus the ID of the pawn which was promoted. For example, if the pawn WP1 is promoted to a Queen, its new name would be CWQ8 (8 being the ID of the WP1 piece).
- More than one piece having a chance to qualify for candidate solution. Generally, when there is more than one candidate to move to a certain square, the PGN notation is required to provide additional annotation to avoid ambiguity. The translating algorithm takes that into account and

translates this additional characters to column-support and row-support variables raising a Boolean flag called support. When the candidates are evaluated these support values work in favour of the correct candidate piece. Method called searchBattle() compares the candidate pieces on several iterations against different criteria to identify which one is the best. The criteria includes validation checks such as:

- Are both pieces active? - every piece has a method isDead() which specifies whether the piece is still active in the game or it has already been taken.
- Are both pieces allowed to make such a move? - this query requires a method defined in the class Board called clearPath() to check if there is a valid path from Square-From to Square-To. This method is designed to account for many little details such as: are the pawns moving in the right direction, is the destination square occupied by a friendly or an enemy piece, are there obstacle pieces in the way, etc.

The searchBatle() evaluation is performed once between two possible pieces from the same type, attempting to qualify for best candidate to move to the destination square, and then the winner is repeating the whole process in a tournament-like battle with possible promoted pieces. The returned best candidate piece, together with the home and destination squares are then forwarded for updating the board data structure and the method for drawing the board is called.

3.3 Drawing the artificial board.

One of the important steps in developing the simulator was to establish to what extent it is reasonable to reflect the real world environment. My strategy in regards to this question was to use the simulator mainly to generate very similar images to the real screenshots taken from the camera, adding some noise and evaluating the algorithm on large number of moves. For creating the artificial image of the board I have used GIMP[2] to extract images of all pieces and an empty board from a real screenshot taken from the camera during the day. Below is given a short summary of the simulators strengths and limitations:

The models of the pieces and the board are directly taken from a real image attempting to reproduce the issues of pieces hiding behind other pieces and the angle of the camera. The square centers have been laid out with different offset values to add displacement of the pieces typical for a real game. These tests have been used to identify good value for one of the important program variables - the threshold of activation of a square if a pixel fells into the range of center coordinates plus threshold. Currently this value is set to 7 pixels as it shows stable behaviour over the whole testing data and does not add additional noise by including more candidate squares. Setting this value properly is a matter of fine calibration as if it is lower than necessary it will miss important candidate

solutions and eventually the algorithm will fail to provide correct translation. Similarly, if the value is higher than necessary, extra squares would be allowed to compete for candidate solutions and this will decrease the accuracy of the algorithm on several occasions. As calibrating this value with real data can be extremely time consuming and even impossible given the need of thousands of analysed moves, using the simulator for this tasks was my only option. The simulator does not attempt to simulate noise typical for real data vision processing such as changes of random pixels due to lighting, shadows, etc. Those type of tests are performed with real data only and they would be described later in the report in the section for Real data and more specifically visual processing.

3.4 Generating images ready for further processing.

The images required for drawing the simulated board are located in folder images. Initially they are read and loaded in the program memory. Then, on every move all pieces are drawn on an empty board image according to the updated board data and merged to form a new image. Every two such images - the one representing the old state of the board and the one representing the change - are forwarded to the move detection algorithm for analysis and once ready the method producing the annotation is called.

3.5 Tests and result analysis.

The final tests are performed over 54 real game records containing about 3680 moves in total.

==========
RESULTS SUMMARY: Completed for 3349 seconds.
54 games have been analysed!

Overall accuracy: 99.01149713324652%

Overall First-fault position: 52.24074074074074

Overall First-fault accuracy: 54.512359330689726%
==========
Number of games that scored 100% accuracy is: 30
==========

Total number of moves analysed is: 3680.0
Average moves per game: 68.14814814814815

148 errors detected...

This is a typical summary of the test results.

3.5.1 Overall Accuracy.

This metric is indicates whether the translated PGN notation is matching the original notation given as an input to the simulator. This metric is important for measuring the improvement of the algorithm but in terms of stability more important is the first fault position metric.

3.5.2 Overall first fault position.

According to this data set the average first fault position for all 3680 games is around the 52nd move of the game out of 68 total moves on average. This counts white and black moves separately. This shows that the algorithm is very stable and normally failing near the end of the game.

3.5.3 Completely accurate translations.

The number of completely accurate translations according to the results are 30 out of 54 games. This is a very good score gives some confirms that the model based approach can be very stable in a closed world environment. With such a good performance on the Simulator, if the algorithm is able to filter good quality images for input of the algorithm, similar results can be expected.

3.5.4 Error analysis.

The algorithm is also counting and recording errors so they can be analysed separately. Here is the list of errors from this test:

GAME NUMBER: 4 strict mode? - false

Error Move N: 28 [correct - wrong] : Rh6 - Rh7

GAME NUMBER: 7 strict mode? - false

Error Move N: 32 [correct - wrong] : Qe7 - Rh5

GAME NUMBER: 9 strict mode? - false

Error Move N: 8 [correct - wrong] : Qd6 - e6

GAME NUMBER: 10 strict mode? - false

Error Move N: 34 [correct - wrong] : Qb3 - d2

GAME NUMBER: 14 strict mode? - false

Error Move N: 35 [correct - wrong] : Rd3+ - Rxd4

GAME NUMBER: 16 strict mode? - false

Error Move N: 9 [correct - wrong] : Ba3 - Qa3

Error Move N: 22 [correct - wrong] : Qa5 - Qa4

Error Move N: 34 [correct - wrong] : Rxh6 - Rh5

GAME NUMBER: 21 strict mode? - false

Error Move N: 20 [correct - wrong] : Qf6+ - c4

Error Move N: 23 [correct - wrong] : Rxc8+ - Rc7

GAME NUMBER: 25 strict mode? - false

Error Move N: 45 [correct - wrong] : Kc6 - d6

GAME NUMBER: 26 strict mode? - false

Error Move N: 30 [correct - wrong] : Kc6 - d6

GAME NUMBER: 27 strict mode? - false

Error Move N: 21 [correct - wrong] : Bb4 - Qb4

GAME NUMBER: 29 strict mode? - false

Error Move N: 41 [correct - wrong] : Rd4+ - Rd5

Error Move N: 47 [correct - wrong] : Qc6 - Qc7

GAME NUMBER: 30 strict mode? - false

Error Move N: 38 [correct - wrong] : Rd6+ - Rd4

Error Move N: 39 [correct - wrong] : Qb6+ - Qb7+

Error Move N: 40 [correct - wrong] : Rd8 - Rd7

GAME NUMBER: 31 strict mode? - false

Error Move N: 11 [correct - wrong] : Qh5+ - Qh6

Error Move N: 13 [correct - wrong] : Qc6 - d6

GAME NUMBER: 32 strict mode? - false

Error Move N: 16 [correct - wrong] : Qh5 - g5

GAME NUMBER: 33 strict mode? - false

Error Move N: 16 [correct - wrong] : Bb7 - Kf5

GAME NUMBER: 34 strict mode? - false

Error Move N: 8 [correct - wrong] : bxc3 - Bb3

GAME NUMBER: 37 strict mode? - false

Error Move N: 34 [correct - wrong] : Ra5 - Ra6

Error Move N: 37 [correct - wrong] : Rc8 - Rc7

GAME NUMBER: 38 strict mode? - false

GAME NUMBER: 39 strict mode? - false

Error Move N: 23 [correct - wrong] : Qh5 - Qh6

Error Move N: 23 [correct - wrong] : Qb8+ - Qb7

Error Move N: 45 [correct - wrong] : Qf4+ - h3

Error Move N: 50 [correct - wrong] : Qh5+ - d2

Error Move N: 54 [correct - wrong] : Qh5+ - d2

Error Move N: 59 [correct - wrong] : Qa6+ - Qa7

GAME NUMBER: 42 strict mode? - false

Error Move N: 10 [correct - wrong] : fxe3 - Nf3

GAME NUMBER: 43 strict mode? - false

Error Move N: 22 [correct - wrong] : Qe7 - Qe6

Error Move N: 23 [correct - wrong] : Qd7 - Qd6

GAME NUMBER: 44 strict mode? - false

Error Move N: 15 [correct - wrong] : Bb6 - d5

Error Move N: 29 [correct - wrong] : Bxf2 - Qxf2+

GAME NUMBER: 45 strict mode? - false

Error Move N: 10 [correct - wrong] : Qf3 - Bd2

GAME NUMBER: 49 strict mode? - false

Error Move N: 30 [correct - wrong] : Qa8+ - Qa7

Error Move N: 34 [correct - wrong] : Rf4 - Rf5

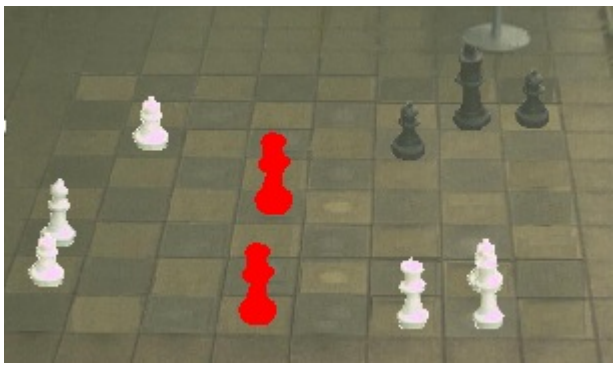
GAME NUMBER: 52 strict mode? - false

Error Move N: 18 [correct - wrong] : Qxg7 - Qg6

GAME NUMBER: 53 strict mode? - false

==========

As it can be seen most of the errors happen with Queens or Rooks. This is so due to the fact that when a Rook or Queen are moving only on the same column, and more specifically to the top of the board, the shadow they cast on their own

example.jpg 

Example 1. Example of ambiguity in analysing moves on the same column for the Queen.

candidate-To solutions makes it very hard to decide which is the best candidate to move to.

A method called Precision Steps for Rook and Queen is implemented to improve the prediction accuracy of the algorithm in such situations. This was the most prominent type of error identified on the tests for the past two weeks. By constant improvement of the little constraints typical for certain part of the board, and certain number of candidate solutions as well as their position relevant to one another, the algorithm improved its accuracy from 95% to 99% and there are high hopes that with sufficient amount of extra testing the model can be improved to reach 100% accuracy.

The latest improvement I have implemented is counting the pixels contributing to each candidate squares so I can use extended reasoning regarding the relevance of certain squares to the actual move.

Chapter 4

Real data mode. Visual processing.

This part of the project relies on functionality built with the help of the simulator for detecting moves and annotating the game but it also introduces a whole new dimension of problems. Working with real data is far more unpredictable and all the issues mentioned above related to noise introduced by the vision need to be addressed at this stage. My application was designed with the assumption that achieving high accuracy on the tests with the simulator combined with obtaining clear images from the camera can be the key to a very solid and efficient algorithm. It turned out that this assumption is not far for truth and in normal weather conditions with enough lighting the algorithm performs very well on real data as well.

4.1 Taking good images from the camera.

The key to high accuracy in terms of the vision part is having as less noise as possible in every subsequent image. Typically, the noise effects are strongly related to the time it passes from one frame to another. Even the smallest changes in lighting, combined with reflections, wind or particles of dust would cause many pixels to deviate from their previous RGB colour values. This project did not have a goal to investigate or measure these noise effects so I havent performed any formal tests to design my algorithm according to the results. Instead, my intention was to use a threshold value for the acceptable range of deviation of the RGB colour values of each pixel and come up with a method for its automatic adjustment according to different levels of intensity of the noise factors. This method is discussed below in the section Automatic adaption of the threshold value.

4.2 Visual pre-processing.

In the context of this project I am referring visual to pre-processing mainly as a procedure of analysing the input of the camera and deciding whether it should pass it forward to the application or discard it as corrupted data. The visual processing algorithms would always attempt to fix the image by manipulating the threshold value but only according to some limitations. These limitations have empirically proven to work satisfactory and their main goal is to provide balance between delay due to image processing and quality of the result. For the moment if every move is made in an interval of 10 to 30 seconds, the algorithm allows enough time for visual processing. If moves are made in longer intervals some of these limitations may have to be changed to adapt to longer games. Examples of corrupted data might be one of the following few cases:

A human player who is still on the chessboard when the image is taken. The player adds huge amount of noise that cannot be reduced even with severe changes of the threshold value. Also, the amount of changed pixels would exceed the allowed limit which is based on estimation of the average number of changed pixels per move. The algorithm will discard such image after unsuccessful attempt to remove the noise to values within the predefined limits. Severe rain, snow or fog. Extreme heat can also be a problem but definitely not one which is easy to test and account for in Edinburgh.

These were examples of added noise but corrupted images may also contain less changed pixels than necessary to detect a move. Such problems, though, are usually found at later stage and are part more of the automatic threshold adaptation rather than the pre-processing.

4.3 Image processing.

This part of the algorithm provides the input for the squareSearcher() method which is essentially where most of the move detection analysis are done. The main task of the image processing method is to compare the input pictures for changes in their RGB color values and identify the regions of change. For work with the simulator no threshold value is allowed and the color values are directly compared for equality, where in the case of real data mode a variable threshold value is enforced which specifies a wide range around the target color value.

In more detail, every two corresponding pixels from the two input images are compared and in case of significant difference between them, they are colored in red. This is done for visual purposes only as the image showing the change region is forwarded to the second Image Panel on the GUI.

Moreover, every colored pixel has to pass another test which checks if it is within the range of any square centre. In this case the affected square is added to an array called search allowing no duplicate elements. Search is collecting the input

for the move detection algorithm. Note that the range around a square centre is defined by one of the important program variables - Point sensitivity, defined as a threshold value in class PointOld. Just for a reminder this value is currently set to 7 pixels and has been derived by experiments both over simulated and real data.

In Simulator mode the Image processing step is called only once and this is sufficient to initialise the search squares as no noise can affect the input images.

In Real data mode, though, the Image Processing procedure may have to be repeated many times if that is required by the pre-processing method, the automatic adaptation of the threshold, or the move detection (more specifically squareSearcher() method) itself in case of insufficient data to reach a solution.

4.4 Automatic adaptation of the threshold for in-range RGB colour values of pixels.

The colour threshold value has a significant impact on the work of the move detection algorithm. It can be expected that the performance on real data would be close to the performance on simulated data only if the image can be cleared from noise to an extend identical to simulated image. A good method of identifying such similarity is counting the number of changed pixels and calculating the average number of changed pixels over the whole game. Such experiments in Simulator mode are not strictly necessary but they give a rough idea about good values that can be desired after noise reduction on real image processing.

Typical values for average changed pixels per game are:

Simulator mode: between 600 - 800 pixels changed on average every move. Real data mode: around 1000 pixels changed on average every move. This data is very useful observation for designing the bounds of the automatic threshold adaptation method. The idea is to set limits around the average value which is updated on every move depending on the previous iterations of the algorithm. In cases where the changed pixels break the restrictions, the threshold value is increased or decreased depending on what boundary had been broken. The two cases and their cause and effects can be summarised in the following way:

The number of changed pixels goes below the lower limit of the average value - the colou threshold is decreased which allows more pixels to contribute to the change area and possibly trigger new search squares. The image becomes more noisy but at the same time more potent in terms of finding new solutions. The number of changed pixels goes beyond the upper limit of the average value - this would increase the colour threshold in order to reduce the number of changed pixels. This procedure will provide clearer image but if not adjusted properly may cause an important search square to be removed from the search array.

The actual default value for the Colour threshold defined in my application is

currently set to 1550000, and in case it needs updating it is either increased or decreased by 200000. Again, these values are derived empirically and attempt to provide a good balance between accurate adjustment of the value and delay caused by threshold updates and re-evaluation.

Normally, the move detection algorithm needs between 2 to 9 search squares to work properly. Having that in mind, gives one more condition to the automatic adaptation of the colour threshold value. If there are not enough squares to compute a solution, it gets decreased to allow more candidate solutions. On the other hand, if there are too many squares, they just act as noise for the move detector algorithm and clearly are a consequence of noise in the image, so by increasing the colour threshold, their number is normally reduced to fit within the expected boundaries.

4.5 Testing.

Testing with real data can be a bit harder that testing using the simulator as more attention has to be paid to the output annotation. The good thing is that one wrong translation is usually easily detectable in a few moves as it causes more errors. If a testing real game is played according to an already recorded game, it can be imported as PGN and checked automatically but otherwise the actual tester is usually a human. Anyway, given that the algorithm doesnt fail completely during the whole game it is very likely that its accuracy would be near 100% unless even if it fails in the last few moves.

I have tested my algorithm on a couple of real games and I have managed to record a game for testing the threshold adaptation. It contains 112 images taken between 4pm - 6pm so that the lighting changes quite significantly and the weather conditions as well in terms of some slight snowing. Using the methods described above my algorithm manages to annotate the game with nearly 100% accuracy.

Here is the transcript of the game I have acquired evaluating the algorithm. It can be checked against the pictures contained in folder exportedPgn or re-played on the application but unfortunately there is no quick way to validate tests done on real data.

1. e4 e5 2. Nf3 Nc6 3. Bb5 Nf6 4. O-O Ng4 5. h3 h5 6. hxg4 hxg4 7. Bxc6 gxf3 8. Qxf3 Qh4 9. Qh3 dxc6 10. Qxh4 Rxh4 11. d3 Bc5 12. Nc3 Bg4 13. Be3 Bb4 14. Na4 O-O-O 15. a3 Rh8 16. f3 Rh1+ 17. Kf2 Rxf1+ 18. Kxf1 Rh1+ 19. Ke2 Rxa1 20. fxg4 Ba5 21. Bxa7 Rb1 22. b4 Bb6 23. Bxb6 cxb6 24. Nxb6+ Kc7 25. Nc4 f6 26. g5 Rc1 27. Kd2 Rg1 28. Ne3 fxg5 29. Ke2 Ra1 30. Ng4 Rxa3 31. Nxe5 Ra4 32. c3 Ra2+ 33. Kf3 Kd6 34. Nc4+ Ke6 35. g4 Rc2 36. Ne3 Rxc3 37. Ke2 b6 38. Kd2 Rb3 39. b5 c5 40. Kc2 Rxb5 41. Kc3 Ke5 42. Nc2 Rb1 43. d4 cxd4 44. Nxd4 Kxe4 45. Ne6 b5 46. Nxg5+ Kf4 47. Ne6+ Kxg4 48. Nxg7 Rc1+ 49. Kb4 Rb1+ 50. Kc5 b4 51. Kc4 b3 52. Kc3 Kf3 53. Ne6 Ke2 54. Nd4+ Kd1 55. Nxb3 Rxb3+ 56. Kxb3

4.6 Homography transformation.

In order to improve the view of the chessboard I have implemented a homography transformation of the board which is calculated automatically and shown on Image Panel 3. Tests on the homography image showed that it brings no further improvement to the work of the model-based algorithm but it could turn useful in future if more visual techniques are applied in order to approach the problem from a different perspective. It sometimes helps to see better certain situations on the board so I have decided to keep using this functionality even only for aesthetic reasons.

Chapter 5

Move detection.

The move detection algorithm compares two snapshots for changes in their pixel values. Then it attempts to identify the two prominent areas of change and triggers model-based search for determining which piece has moved where.

In more detail, it compares the whole image pixel by pixel and record the changes within a certain threshold. Due to noise in the image, there are usually more than two main regions of change but the goal is to narrow them down to two - where the piece has moved from and where the piece has moved to. The approach I am using is to relate the area of change to a particular set of squares by computing the distance between their centres and the affected pixels.

In Simulator mode as the images are generated artificially there is no need for further image processing to improve its quality. So in this case there is no need for allowing threshold value other than 0 for the pixel color comparison.

Real data move detection requires the two pictures to have some thresholding for their comparison in order to remove noise and analyse only the regions affected by the movement of the pieces on the board.

5.1 Design and implementation of the algorithm.

Undoubtedly the most difficult part of the project that also contains the core functionality of the application was the design and implementation of the move detection algorithm. My approach to building this complex logical mechanism was to design the main possible branches and then use the simulator to test for rare situations and attempt to extend the common cases so that they can adapt to every of the rare cases. This process is extremely difficult as adapting the algorithm to a certain set of conditions increases the accuracy of the target type of data, but may become biased to it and treat it with priority. Finding the optimal balance turned to be a problem with surprisingly high complexity. At some point my work on the project acquired a new goal that was interesting enough to investigate - researching whether such an algorithm can be designed to achieve

100% accuracy in a closed environment such as the simulator games where no external influence can be blamed for the fail of the algorithm. From all the discussed alternatives before it seemed that if this idea is possible, delivering such algorithm combined with techniques for adapting the vision processing so that it can provide clear images for it would be a very good and stable general solution to the problem of annotating a chess game in such conditions. Although, I have reached accuracy near 100%, it can not be guaranteed that new games would not introduce problems that are difficult to handle with the current methods and the program needs to be improved again. The issue is not only with the impossibility to prove the claim that every single possible move had been accounted for, but moreover, modifying the algorithm to work with new moves better doesnt guarantee that it will still perform well on the moves from the old data sets.

With a few minor differences the move detection algorithm works the same way in Simulator and Real data modes. Although it was easier to separate them and use two different algorithms the decision to both modes running with one algorithm is an intentional design decision based on the fact that improving the closed world algorithm using the simulator would improve the performance on real data as well.

I have already explained some of the steps necessary for the initiation of the move detection algorithm in previous chapters but I am going to summarise them concisely in order to illustrate the structure of the system from the perspective of the move detection part.

Recall that when two images are first passed to the moving detector algorithm they are compared for changes in their RGB color values for each pixel and the squares affected by this change form the input set for the future evaluations. It is important to note that another array called amount stores the number of pixels within range of a certain square centre and in that way acts as a measure of intensity for how strongly a square is affected by the change. Normally, squares who are seriously affected by the move would have higher amount values but this is not always true due to pieces that might be hiding the actual active spots.

The logical beginning of the move detection algorithm can be identified as the call to method squareSearcher() in class Move Detector. It requires no arguments as the data it works with is freely accessible within the class - an array containing all squares that have potentially been affected by the change in pixel values (see Image Processing chapter).

Here is a quick outline of the squareSearcher() method that will be discussed in more detail in this section:

Method name: squareSearcher()
Home class: chess.board.MoveDetector.
Input data: Array of squares affected by the pixel color change.
Output data: A pair of most probable candidate solutions Square-From and Square-To referring to the square where the piece was and the square to which it has now moved.

Main task: To perform logical analysis and reasoning capable of narrowing down the set of affected squares to a single candidate pair.
Sub-task: Set flags and program variables to assist with the correct translation of the data structures to PGN notation.

5.2 Inner workings.

The squareSearcher() algorithm exercises logical control over smaller methods dealing with specific tasks in the process of finding the candidate pair squares. I am going to attempt to provide a detailed overview of the inner workings of the algorithm in this section. The sqareSearcher() algorithm works mainly with data which is accessible within the class. Its primary goal is to find the candidate solutions for the current move but it plays just as important role in setting flags and program variables that are later on going to be used for the correct translation of the moves into PGN notation. The following sequence of stages illustrates the procedural execution of the algorithm.

(a) Split the search array into smaller arrays.

Divide and conquer technique for gaining more control over the input data. By checking whether each square element contains a piece or not, the search array is split to two sub-arrays:

Candidates-From: Contains only squares from search which have a piece on them, having the same color as the color of the player whose turn it is. The boolean variable isWhite is passed to the MoveDetector class every time the Simulator or RealData classes attempt to execute one of its methods. This efficiently narrows down the number of candidate solutions for finding Square-From. Candidates-To: In order to capture all possible solutions, this array receives all empty candidate squares and the squares containing a piece from the opposite color. It is meaningless to store pieces from the same color as according to the rules of the game they can not be a destination square.

After the initialization of these two new arrays the algorithm is ready to proceed with further manipulations in order to narrow down the data even further to the most probable solution.

There are a number of special moves and chess situations that I am going to discuss in a separate section of this report. For the moment as I am explaining the overall structure and order of execution of the squareSearcher() method I will only reference them briefly. These special moves are handled in separate methods that check for certain activation conditions on each move.

(b) Handle castle.

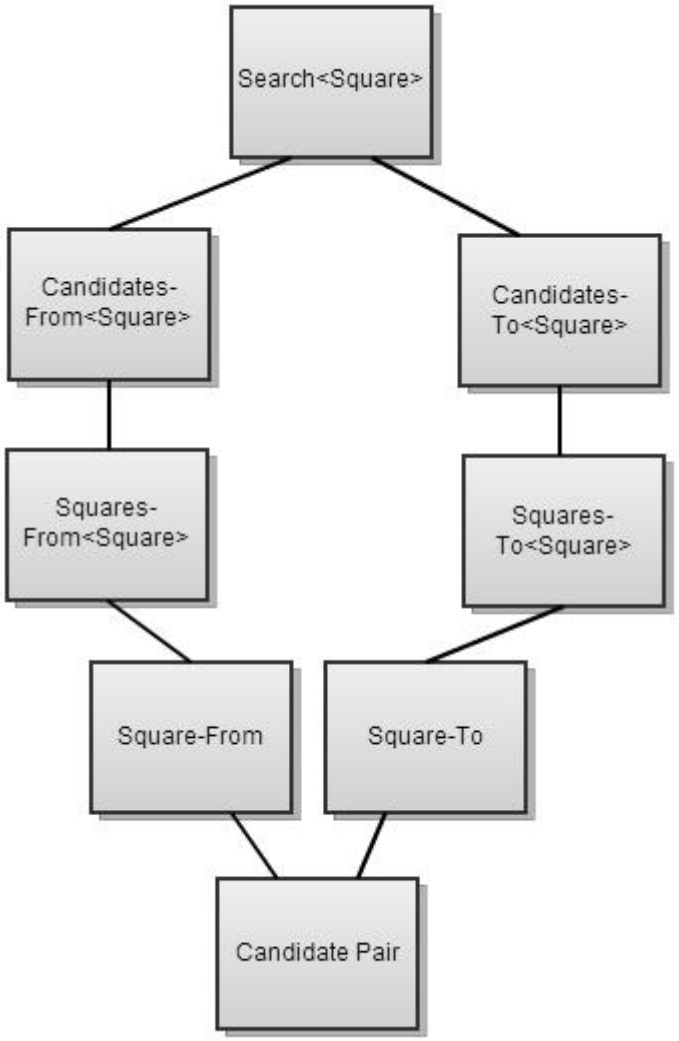


Figure 3. Organising input arrays data to reach a single candidate solution.

The first such sub-method is handling the castle situation. It is important that this method is the first one to be executed as the castle situation is quite different than other standards moves as the rules by which the king and the castle piece change their positions are not consistent with the general rules for moving a piece. In order words, I am treating castling as a highly exceptional move which in fact can occur only once per game for each player. Identifying and handling the castle situation correctly requires no further search for candidate pair solution, which is another reason I prefer to start with it.

(c) Populate Squares-From and Squares-To.

The next step in the execution of squareSearcher() is manipulating the data in the Candidates-From and Candidates-To arrays to produce Squares-From and Sqares-To arrays and the first candidate pair Square-From and Square-To.

Intuitively, in order to reduce the set of candidate squares to a more certain set of solutions some strong constraint have to be enforced - can a piece legally move from any of the Candidates-From to one of the Candidates-To square. If this condition is satisfied the squares are transferred to the new arrays Sqares-From and Squares-To.

This legal move evaluation is implemented by two very important methods that are used at many places in the program and I would like to introduce here, explaining the differences between them:

- canMoveTo() - this method is defined in class Piece and takes a square object as an input. This is purely mathematical computation returning true if the piece can theoretically make a move to the input square and false - if such move would be illegal for the given piece. I have derived the formulas for each piece on my own but they appear to work correctly for the methods I have implemented. I am going to provide a short description of my formulas for each piece. The translation of the formulas would be as general as possible but all the variables would be defined similarly to the setting in the code. Note that this was a method implemented in the Piece class so it has easy access to the whole information about the given piece and it also has the target square passed as an argument.
  - Definition of variables for the logical formulas: Row - the row where the piece currently is. Col - the column there the piece currently is. RowTo - the target row. ColTo- the target column. abs—x— - the absolute value of x.
  - Knights formula: (abs—row - rowTo— = 2 AND abs—col - colTo— = 1) OR (abs—row - rowTo— = 1 AND abs—col - colTo— = 2)
  - Bishops formula: abs—row - rowTo— = abs—col - colTo—
  - Castles formula: row = rowTo OR col == colTo

  - Queens formula (composition of the Castles and the Bishops formulas): (row = rowTo OR col == colT) OR (abs—row - rowTo— = abs—col - colTo)
  - Kings formula: (abs—row - rowTo— = 1 AND abs—col - colTo— = 0) OR (abs—row - rowTo— = 0 AND.abs—col - colTo— = 1) OR (abs—row - rowTo— = 1 AND abs—col - colTo— = 1) OR
  - Pawns formula: abs—row - rowTo— = 1 - for standard move abs—row - rowTo— = 2 - possible for the first move only. abs—row - rowTo— = 1 AND abs—col - colTo— = 1 - in case of pawn take.

- clearPath() - defined in class Board, this method performs heavier computations. It uses canMoveTo() as well as several other methods to check for clear path between a starting and a destination square. If the canMoveTo() method returns positive result and the target square is either empty or occupied by an enemy piece a step by step traversal of the path between the home and the destination square is performed and every square on the way is checked for pieces on them. The algorithm would return positive value only if there are no obstacle pieces on the way between the home and the target square. This method best extends the Board class due to several reasons:
  - it needs data about the whole board both in terms of Square and Piece objects;
  - it is very general and has a big application in the whole program and the Board class has instances almost everywhere.

(d) Candidate Pairs tree data structure.

From the initial stages of the project I had the idea of storying the game data in a tree structure in order to keep multiple hypothesis and check them for consistency at later moves. It seemed like a very powerful approach that can resolve the uncertainties caused by restrictions in the vision processing or the model-based move detector algorithm. For the most part of the project I was working on this idea and although I have decided to abandon it completely for the moment, I believe it can have positive impact and it definitely worth mentioning.

The first strategy was to have a tree data structure which on every move is adding several new candidate solutions. On later moves they are checked for consistency and branches who fail the checks are removed. Eventually, the plan is to deliver one or few alternative annotations of the game with the hope that at least one of them would represent the true history of the game.

On later stages I have optimised the idea to use as little memory as possible and part of the functionality of this data structure is still in use by the algorithm. Rather than storing the whole board data in a tree, which is costly and in fact very restricted operation in Java, I am working with a very small

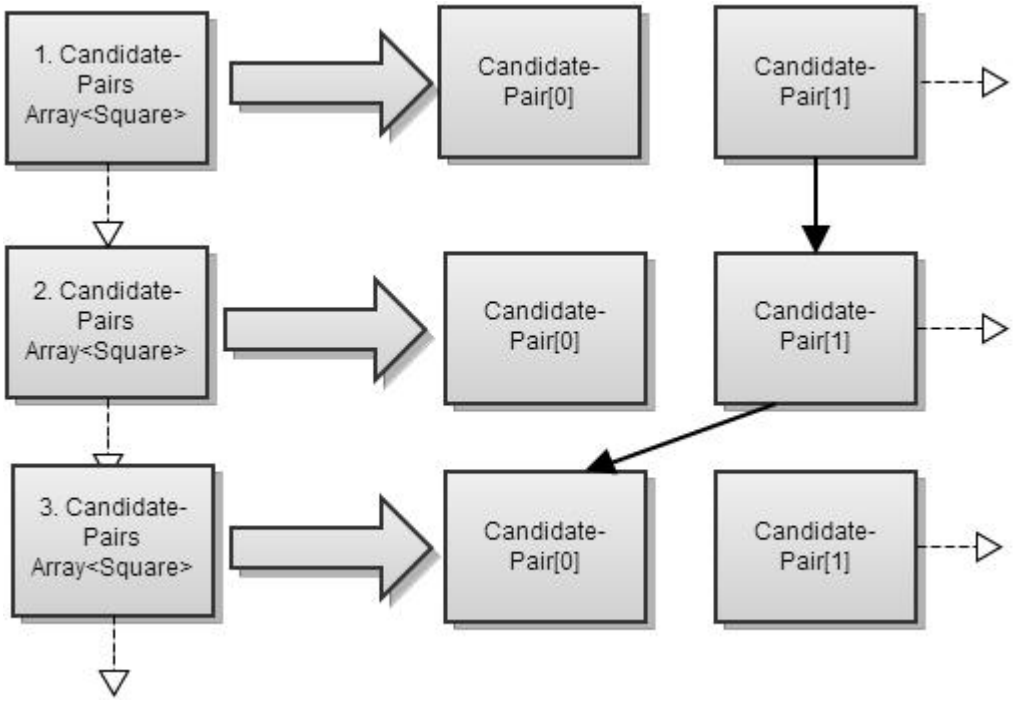


Diagram 2. Hierarchical dependency of candidate pairs .

subset of the data contained in the object CandidateTuple. This class has Square-From, Square-To which are instances of class Square and a field child - an instance of CandudateTuple. An array list holds all candidate tuples per move and they can keep parent-child dependencies between them (check Figure 3. for example). Figure 3. Candidate Pair tree dependency. The black arrows represent parent-child dependency.

On the above picture, if a parent becomes inconsistent for example, a chain reaction disables all its children as well.

The mechanics of this design worked quite nicely but my problem with this approach came from strategical point of view rather than technical. It seems that keeping such data structure active would be very useful for algorithm based mainly on visual processing as it improves its chances to fight against noise in the vision data or other kind of uncertainty situations. The honest description of the effects of this method on a model-based approach algorithm are not very positive though - it simply spoils the discipline of the algorithm and I will try to explain what I mean by that. Typically a model-based algorithm uses the logic of chess and the current state of the data to reason about what move could possibly have been made given the input squares. Built with the help of simulating many different scenarios, my algorithm attempts to go as deep as possible in the low level of different cases, distinguish their unique logical patterns and treat them differently. The advantage of successfully identifying and separating a situation from others is that once evaluated correctly it will always be in future under the same conditions. The problem of using the tree data structure is that it acts as giving up to fight for a solution. If the algorithm is toler-

ated to stop at some point and admit that it cannot distinguish the correct candidate from several, what can guarantee that on the next iteration it can perform any better? My observations while testing this theory showed that the number of unresolved branches kept growing as the same functionality that caused the first branch to emerge repeatedly manifested in its children causing quick expansion. Another problem was that the conditions for inconsistency are hard to be defined and insufficient, especially in a later game when there are fewer pieces on the board. As the algorithm always makes the assumption that the data so far was correct and computes its new assumptions using it, the only inconsistency can be caused only if a piece attempts to move to a square held by another piece from the same colour. Also, this approach never leaves the algorithm without any solutions and thus it is hard to trigger procedures such as automatic color threshold adaptation and other helpful techniques that can be used to reach a single best solution.

(e) Precision steps.

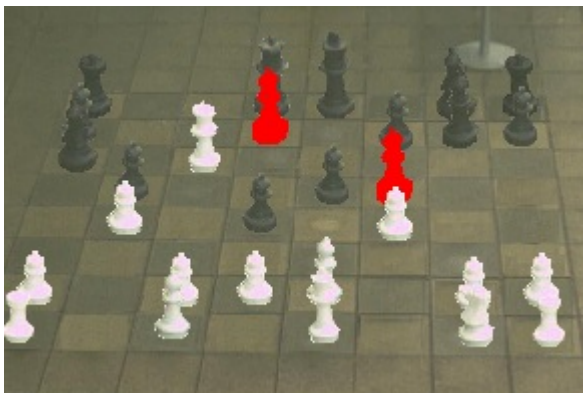
With this term I am referring to a series of methods that are allowed to modify the best candidate pair in attempt to improve the chances of the correct solutions to take their place. Designing and testing precision steps can be very hard and time consuming as a new improvement could be considered working and beneficial to the project only after hours of testing over the whole data set.

The precision steps can be split into several groups according to their targets:

- General - they are enforced every time the candidate pair falls under certain conditions no matter what piece is occupying the squares.
- Piece specific - the conditions of the precision step are enforced only if a certain piece is a part of the candidate pair.
- Situation specific - invoked only when some specific situation occurs. Those type of situations are just subsets of standard chess moves and are not related to the special situations discussed below such as castle, pawn promotion, etc.
- Piece and situation specific - a type of very low level precision adjustment that really makes the difference between 95% and 100% accuracy. Incredibly complex to design as they should satisfy the constraints of hundreds of moves at a very low level and direct each situation to an appropriate solution.

After this brief overview of the terminology I am using to describe these precision enhancing methods I am going to describe the motivation and use of some of the important ones in my program.

i. Square-From correction - this precision step is based on the assumption that the most probable candidate starting square is usually the one



Example 1. Queen - Bishop diagonal movement.

on the lowest row in the data set. This is usually safe to assume because of several reasons:

Pieces cast shadow and usually affect the squares above them so the squares usually involved in the actual move are usually the ones located at the lowest row. There are very rare situations where this would not be the case if any and they have very high probability to be fixed by the other precision steps.

ii. Diagonal movement correction - this is a delicate situation which occurs when a piece is casting a shadow on the pieces behind due to its movement and their movement is also a valid move. For example Queen in front of a Bishop moving diagonally to a certain square would cause ambiguity as it immediately creates a second possible candidate pair - the Bishops Square-From and Square-To which happens to be the square above the Queens Square-To. My approach for avoiding such ambiguity is similar to the assumption made in the discussion of the previous situation - lower row pieces are more likely to be the initiators of the pixel change. In this situation, though, the method should compare the complete candidate pair rather than just the squares individually because it might be the case that the Squares-From of one of the pairs is matching with the Squares-To from the other.

Usually, the diagonal movement correction is done near the end of the squareSearcher() algorithm when the Candidate Pair data structures are filled with all the candidates pairs for this move. Then the solution to the problem is to iterate through the array of pairs and find the one with minimal sum of its Square-From and Square-To row values. This simple operation ensures that the correct pair is selected.

iii. Queen and Rook correction.

This precision step is one of the hardest to manage not only because of the high mobility of these pieces but also due to the fact that they may cast long shadows that appear longer or shorter depending on

Example 2. Example of ambiguity in analysing moves on the same column.

which half of the board they are occupying. Consider the situation when in late game the Queen is on A5, there are no other pieces on the A column, and the Candidate-To Squares are: A6, A7, A8 due to the shadow the Queen casts on the board. In such situation it is almost impossible to tell where is the correct Square-To. Similar is the situation with the Rook pieces. Nevertheless, I have started work on a sufficiently large set of precision steps dealing with these situations. I had to implement additional amount counter for each Search square indicating how many pixels contribute to its promotion. In that way I am hoping to reduce the search space by removing pieces which are only remotely touched by the tops of the Queen or Rook pieces. This is an example of very hard situation to manage, as every little change in the precision steps can affect a large proportion of the moves and affect the accuracy of the algorithm significantly - rooks and queens are generating a big portion of the moves per game.

5.3 Special situations.

In the process of logical analysis of the data there are several distinctive chess situations that require special attention. Implementing mechanisms to handle them implies that the algorithm would be able to annotate games with very good accuracy due to their presence in almost every chess game.

i. Castle situation.

   The castle situation occurs when the King and one of the Rooks change their locations according to a defined castling rule in the chess game. Usually the King would move two squares to the right - from E to G for small, Kingside Castle, usually annotated as O-O, or two squares to the left from E to C for Queenside Castle, annotated as O-O-O. The Rook would normally take its place on the other side of the King and move from H to F fo Kingside Castle and from A to D for Queenside Castle. Note that castling is not permitted if the King or the Rook have moved from their initial positions during the game even if they

have managed to return to them again. Having that in mind makes accounting for castle citations relatively easy.

In Real data mode it is enough to check if the Candidates-From contain the King and the Rook piece and if the Candidates-To contain the relative destination squares for the castle move. If these conditions are met, then the board data is simply updated and the castle move annotated.

A little bit more complicated is the situation with the Simulator. Due to the fact that the drawing method treats the castling as two separate moves the squareSearcher() method has to wait one move to receive the second part of the complete castling move and then mark it as executed. This creates a slight inconvenience but it is easily manageable by using two boolean flags for castling instead of only one.

ii. Check / Checkmate.

Checking for check or checkmate is a very important operation for every chess engine. For the annotation algorithm it is only a minor detail which adds + or sign at the end of the move transcription.

I have implemented a method chick identifies if the move delivers check and adds the appropriate notation in such case. Check situations are far more frequent than checkmate and a lot cheaper for computing. My strategy is at the end of each move to create artificial piece that mimics the piece that is just going to Square-To. Placing this imaginary piece on Square-To, followed by searching from clear path between Square-To and the Square of the opponent King results precisely in checking if the piece would deliver check upon arrival to the target square. This is typically one of the last checks in a translation algorithm because it adds the + symbol at the end of the annotation string.

I havent implemented checkmate solution yet due to several important reasons:

it is a move that occurs only once per game and only if the game is player until the end and thus has very low significance for the annotation algorithm. it is also hard to implement and requires many computations of interactions between different pieces.

iii. Pawn promotion.

Pawn promotion is another very delicate situation typically annotated in the following way - e8=Q - meaning pawn moves to E8 and gets promoted to a Queen. Every time a pawn reaches the end of the board it gets promoted to a major piece according to the choice of the player: Queen, Knight, Rook or Bishop. The most powerful pick is obviously the Queen but often chess situations require a Knight to be picked to deliver a nice checkmate or some of the other piece to be prefered in

order to avoid stalemate or just go better with the strategy. Having this in mind it is inappropriate to permanently assume one of the options but perfectly reasonable to make a guess and check if it gets confirmed in the following moves. This is the only tactic that seems reasonable for Real data mode. Basically, once a pawn reaches the end of the board it is allowed to make almost any legal move until it settles in the rules specified by its new role.

In simulator mode the things are easier as pawn promotion is specified by the notation and the algorithm can easily check if the right decision is taken.

As the pawn promotion provides the piece with new functionality, that needs to be marked explicitly. As Piece IDs are unique identifiers of each piece I am keeping them unchanged even in a case of pawn promotion. To indicate the promotion I am adding a special character to the name of the piece as well as changing it to represent both the new and the old piece. For example, assume pawn with code name WP2 gets promoted to a Queen. The piece ID remains the same, but the name gets changed to WQC9 - WQ is the typical name for a White Queen, C is the special code character indicating promotion and 9 is the ID of the pawn. I prefer to add the ID of the pawn as well in order to avoid ambiguity in case of more than one Queen promotion. Using this technique, it is guaranteed that enough unique names can be generated even in case of all eight pawns get promoted. Upon translation to PGN, additional character Z is added to the promoted piece to indicate that the promotion has been annotated. The special field promoted in the Piece class is set to true. Finally, an example of a newly promoted piece name would look something like WQC9Z.

iv. En Passant pawn take.

En Passant pawn take is a situation which can occur only if an opponent moves one of its pawns two squares ahead and it lands next to a friendly pawn. Then the player whose turn it is can decide to take that pawn as the square it just jumped over was constantly threatened beforehand. Though, not so hard to get used to that rule in real game, it presents some technical difficulties to the algorithm as it introduces a special condition under which a piece can be taken by not even moving to its square.

For solving this problem efficiently I am performing a check for En Passant pawn take at the end of every move that could possibly trigger one. If the test is positive, some boolean flags are raised and the destination square in case of En Passant take is stored in the program memory for easy access.

5.4 Finding no solutions.

Rather than always continuing providing even a weak candidate pair, the squareSearcher() algorithm is actually able to ask for help by refusing to provide any solution in case of inadequate input data. If the input data set is suspiciously large or small or does not provide enough candidates to qualify for strong candidate solution, it would force the colour threshold value to update itself and the visual processing would be restarted in order to generate new set of candidate squares. This procedure would typically be triggered as well if the size of the input data array is less than two or greater than 9 squares. The whole process is explained in more detail in the Chapter Visual processing - Automatic adaptation of the Colour threshold value.

Chapter 6

Game annotation.

Providing a transcript of the game history is a very important requirement for the project. For producing the annotation I have decided to use the Portable Game Notation (PGN) standard which is easy to export and import in many chess applications.

There are two stages where PGN notation is used in the application. The first is at the initialisation of the Simulator mode where a file containing games recorded in PGN format is imported. Then the games are translated from PGN notation into commands for updating the game data structures.

Similarly at the end of each iteration of the Move Detector algorithm, the game data structures have to be translated into PGN notation. This is done by a composite procedure extending the Simulator or Real Data classes right into the heart of the Move Detector class. It is much easier to translate the game from within the Move Detector class as the squareSearcher() method has already found the most probable candidate pair and different flags indicating special situations have been initialized.

A call to the method translateToPgn() or dataTranslateToPgn() needs no arguments as both method have access to the variables in class Move Detector, begin themselves members of it. Both methods generate and return an output string as the procedure is mimicking the procedural steps of move detection:

Posible castle moves are evaluated first and translated to O-O for Kingside or O-O-O for Queenside castle. Square-From is evaluated and according to the contained piece a piece code character is added to the output. a .. h - for pawns, K for King, Q for Queen, N for Knight, B for Bishop, R for Rook. No color code is necessary as this is specified by the position of the output in the annotation list. If more than one piece from a given type can move to the same square the correct piece is additionally supported with reference to its column or if this is still insufficient - to his row as well. Example: Nc6xd4. A check is made for piece capture - in case Square-To had a piece on it a special character x is added to indicate capture. Square-To is translated to PGN - this is very straight forward step as the names of the squares in memory are equivalent to PGN notation so it just requires cast to lower case of the Square-To name field. En Passant flags are checked and if such situation was detected the e.p code is added to the output indicating the special pawn take. If flags indicating pawn promotion are active

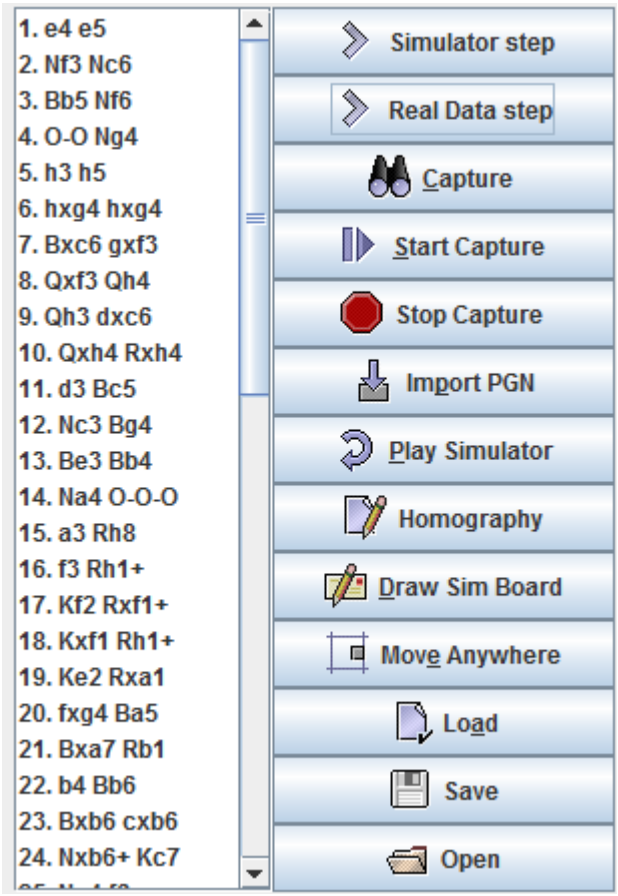


Figure 4. Game annotation printed on the screen.

the equality sign is added to the output as well as the code of the promoted piece: =Q for example. Finally, the translate to algorithm computes the check situation and if the current move delivers a check a + is added to the output. Following this algorithm produces exact representations of the moves in PGN notation: Rf8xf2+, exd4, fxg1=Q, etc.

Chapter 7

Conclusion.

The final tests show very good performance both on simulated and real data. The testing data set is large enough to attract the attention to specific types of errors which can then be eliminated over time by experimenting with better methods.

It should be taken into account that because the iteration over the testing data is done really fast and it is completely automatic, sometimes errors occur because the computer is not fast enough to keep up with drawing the pieces and possibly 1 out of 100 errors at least is due to such behaviour. For the moment, the goal of reaching 100% stability on the simulator looks achievable but the current score is 99,01% is also not a bad achievement.

For the games I have tested on real data, if the algorithm has enough time to adjust its colour threshold, it always provides meaningful analysis. Even if the translation is wrong, the true solution is always contained in one of the candidate pairs.

Due to the complete similarity between the artificially drawn images and the real ones, evaluating the algorithm over large number of moves can provide some solid proof that it works correctly and even generate more formal test analysis. It also provides estimation of the overall accuracy over a given data set and can be used in future for extracting other useful chess statistics.

Implementing the idea of the simulator was very beneficial in terms of improving the performance of the algorithm as well. By using wide variety of games and eventually having some statistics and results to analyse special cases were easy to spot and work on.

I would like to discuss briefly some of the good sides of the project I am personally satisfied about:

- Fulfils the functional requirements of the project.
- Works with real data on automatic, semi-automatic and manual mode.
- 100% own work designed for change and extensibility.
- Intuitive GUI and tools for observation and control.

- Delivering Simulator - tool for testing and validation of the project beyond expectation.
- Functionality for Import/Export to PGN notation - official chess standard.
- Stable model-based algorithm scoring 99,01% accuracy on 54 real games, 3680 moves.

7.1 Future work:

There is not much that is left to be done on this project but it was clearly designed as an application that can be extended to something better so I would like to provide some bullet point suggestions for future work anyway:

- Calibrate the visual processing for longer intervals of image captures - hours, days.
- Add support row/column in the exported PGN notation.
- Improve the vision to assist in ambiguous situations.
- Implement checkmate evaluation.
- Implement an interface for communication to a open source chess engine.
- Re-factor, and redesign the algorithm for iPhone, Android OS to annotate any chess game recorded with the phone camera.

Bibliography

[1] Bob Fisher, Introduction to Vision and robotics lecture notes, 2012.

[2] Bob Fisher, Dilate and Erode, http://homepages.inf.ed.ac.uk/rbf/

[3] Richard Hartley and Andrew Zisserman, Multiple View Geometry in Computer Vision cap. 4

[4] 2D Homography: Algorithm and Tool, http://mmlab.disi.unitn.it/wiki/index.php/2D_Homography:_Algorithm_and_Tool

[5] Steven J. Edwards, Portable Game Notation Specification and Implementation Guide, ICC Help: PGN-spec, http://www6.chessclub.com/help/PGN-spec

[6] B. Majecka, "Statistical models of pedestrian behaviour in the Forum", MSc Dissertation, School of Informatics, University of Edinburgh, 2009.