Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Tomáš Musil

Optical Game Position Recognition in the Board Game of Go

Department of Applied Mathematics

Supervisor of the bachelor thesis:	Mgr. Petr Baudiš
Study programme:	Computer Science
Specialization:	General Computer Science

Prague 2014

I would like to thank Mgr. Petr Baudiš for his supervision, patience and valuable advice. I would also like to thank my parents for their support during my studies.

Název práce: Rozpoznávání pozic deskové hry go z fotografií

Autor: Tomáš Musil

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Petr Baudiš, Katedra aplikované matematiky

Abstrakt: Při profesionálních a turnajových partiích ve hře go bývá obvykle pořizován jejich zápis. I neformální partie může být užitečné si zaznamenat pro potřeby pozdější analýzy a studia. Pořizování záznamu však hráče ruší od hry a snadno se při něm udělá chyba. Videozáznam nebo soubor fotografií postrádá flexibilitu abstraktní notace. V této práci popisujeme možné postupy při automatické extrakci herních pozic z fotografií. Navrhujeme vlastní algoritmus založený na Houghově transformaci a metodě RANSAC. Součástí práce je implementace programu, který tento algoritmus využívá k tomu, aby umožnil hráčům partii snadno a spolehlivě zaznamenat.

Klíčová slova: počítačové vidění, hra go

Title: Optical Game Position Recognition in the Board Game of Go

Author: Tomáš Musil

Department: Department of Applied Mathematics

Supervisor: Mgr. Petr Baudiš, Department of Applied Mathematics

Abstract: It is customary to keep a written game record of professional or high-rank amateur tournament games of Go. Even informal games are worth recording for subsequent analysis. Writing the game record by hand distracts the player from the game and it is not very reliable. Video or photographic record lacks the flexibility of abstract notation. In this thesis we discuss several ways of automatically extracting Go game records from photographs. We propose our own method based on Hough transform and RANSAC paradigm. We implement a reliable and easy to use system that allows players to take a game record effortlessly.

Keywords: computer vision, game of go, baduk

Contents

Introduction

The game of Go

The game of Go is an ancient board game, originating in China. It was first mentioned in Chinese writings from Honan dating from about 625 B. C. (Bell, 1979).

It is traditionally played with black and white stones on wooden board called Goban. There is a square grid on the board, traditionally consisting of 19 × 19 lines. During the game, stones are placed on the intersections. Smaller boards are used for beginners (usually 13 × 13 or 9 × 9 lines).

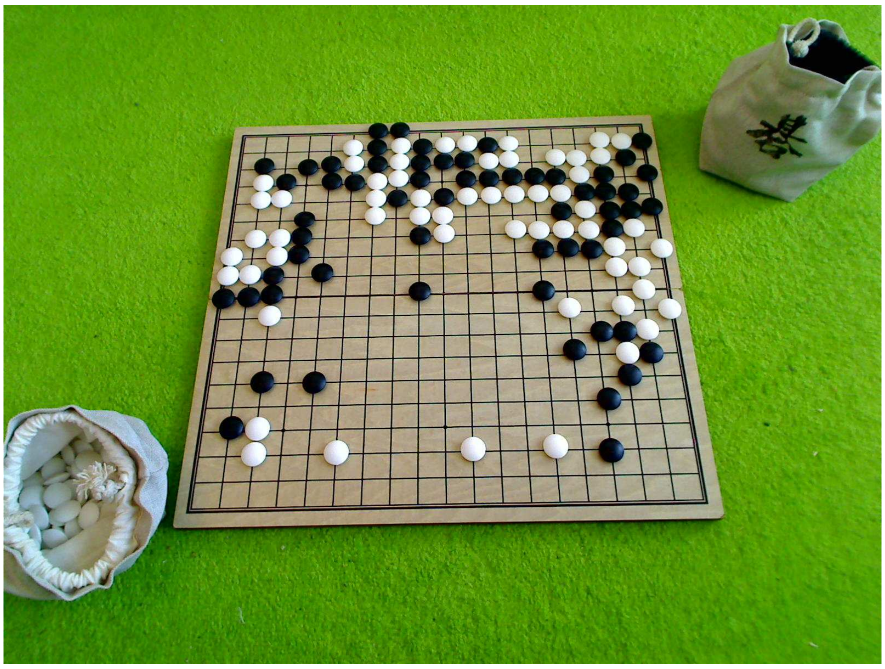

Figure 1: A game of Go in progress.

Two players take turns in placing a stone on the board. Once the stone is put down on the board, it does not move. It can however be removed from the board ("captured"). The goal of the game is to control more territory than the opposing player. Because the rules of the game are not directly relevant to our work, we refer interested reader to literature (e.g. Iwamoto, 1972) for comprehensive description of the rules. The rules can also be found on the internet, for example on *Sensei's Library*[1] or *Wikipedia*[2].

Recording the game

The traditional game record (called *kifu* in Japanese) is handwritten on a special form with a grid. The number of each move is written on the location where it was played. Two pens of different color are usually used — one for each player's

[1] `http://senseis.xmp.net/?RulesOfGo`
[2] `http://en.wikipedia.org/wiki/Rules_of_Go`

moves — to make the game record easier to read. Any mistakes in a handwritten record are hard to correct.

Today, computers, tablets or smartphones can be used to record the game. This facilitates the recording, because mistakes are easily corrected and the resulting game record can be displayed conveniently. But it still distracts the recording player from the game.

Outline of this thesis

In this thesis, we examine possibilities of automatic optical game position recognition in Go. We implement a system that enables players to record their game easily, with just a (web)camera and a computer. The recording is fully automatic and does not distract the player from the game. It is reliable and the resulting game record can be easily displayed and edited in any Go editing program.

In Chapter 1 we introduce basic notions of perspective geometry, color theory and we describe algorithms that are used later on. In Chapter 2 we give a detailed overview of the problem of optical recognition of Go positions. In Chapter 3 we survey related academic works and currently available Go recording software. We describe our method of game position recognition in Chapter 4 and the application based on it in Chapter 5. We evaluate performance of our application in Chapter 6.

1. Theory and algorithms

In this chapter we explain some basic theoretic notions and algorithms that will be used in following chapters.

1.1 Perspective

If we took a picture of a square grid with a camera parallel to the grid and positioned directly above its center, the image would contain a square grid. Because we do not want to impose such conditions on users of our program, we have to deal with the image of a perspectively transformed square grid. Traditional Go boards also take this into consideration, so the grid is in fact rectangular, to appear square for a player sitting next to it and looking at an angle.

When reasoning about perspective transformation, we assume three-dimensional scene to be projected on a two-dimensional plane through a single point (called the *focus point*).

For each set of lines in the scene that are parallel to each other, we have a set of lines in the image that all meet at common point. This point is called the *vanishing point*. When the lines are also parallel to the projection plane, the vanishing point lies in infinity. This is motivation for introducing *projective geometry*, but for our purpose vanishing points in standard vector space are sufficient.

Depending on number of vanishing points in the image we distinguish one- or two- point perspective. Perspective with more vanishing points are common, but since we are only interested in board-grid consisting of two sets of parallel lines, they are irrelevant to us.

Important property of the perspective transform is that it preserves lines. This means that a line in the scene is projected onto a line in the image, the intersection of two lines in the scene is projected onto a intersection of corresponding lines in the image, and so on. We can use this to derive relations such as that if we connect the corners of the board in the image, we get the center of the board at the intersection of diagonals. This can be used to compute all the intersections in the grid just from the coordinates of the corners of the grid in the image.

For more information on the use of perspective in computer vision, we refer the reader to (Sonka et al., 2014).

1.2 Color spaces

A color space is an abstract mathematical model describing a way to represent colors as vectors of numbers. The most common color space for storing and manipulating images is RGB. This is because both computer screens and digital camera sensors use red, green and blue channels for displaying and recording images. Human eye also contains three kinds of cone cells that respond to red, green and blue light.

RGB has no direct semantic interpretation. For image analysis, we might want to use a different color space. Examples of such color spaces include HSL and HSV (Joblove and Greenberg, 1978). These spaces consist of hue, saturation, and lightness/value, which are easier to interpret than RGB values.

To obtain a gray-scale image from a polychromatic one, we use luminance. It comes from the YUV color space and its variants, which are mostly used in video encoding. The luminance is defined as $Y = 0.2126R + 0.7152G + 0.0722B$. This reflects the fact that green light contributes the most to human visual perception of intensity, and blue light the least. Digital cameras usually have better resolution in the green channel, so it makes sense to give the biggest weight to green in image analysis as well.

1.3 Hough transform

The Hough transform for lines was introduced by Duda and Hart (1972), building on a previous work by Hough (1962).

We detect lines in the image by transforming it to the Hough space. A point in the Hough space corresponds to a line in the original image. Each line is represented by its angle and distance from the center of the image. The Hough transform works on gray-scale images. Therefore every point in the image has just one value (we will refer to it as *intensity*). Every point in the Hough space — specified by the two parameters of the corresponding line — has just one value as well. The Hough transform is a simple voting algorithm. Every point P in the image votes by adding its intensity to the intensity of every point in the Hough space that corresponds to a line that P lies on. Consequently, intensity of a point in the Hough space represents cumulative intensity of the corresponding line in the image. Peaks in the Hough space then indicate distinct lines in the image.

Algorithm 1 Hough transform

```
function HOUGH_TRANSFORM(img)
    function DIST((x, y), θ)
        $L$ is the line going through point $[x, y]$ at angle $\theta$
        $d \leftarrow x \cdot cos(\theta) + y \cdot sin(\theta)$        ▷ distance of $L$ from the origin
        return ROUND($d$)
    $m \leftarrow zeroes$
    for $x, y \leftarrow (0, 0), size(img)$ do
        if $img[x, y] > 0$ then
            for $\alpha \leftarrow 0, \pi$ do        ▷ take angles between 0 and $\pi$ with some step
                $d \leftarrow$ DIST($(x, y), \alpha$)
                $m[\alpha, d] \leftarrow m[\alpha, d] + img[x, y]$
    return $m$
```

As described so far, Hough transform would be a continuous function. In practice, we naturally compute only a discrete sample from it. Because the input image is also discrete, we have to reasonably define the relation of lying on a line. We solve this by considering only a subset of angles for lines (uniformly sampled with some step). When computing on which lines does point P lie, we compute the distance d for every angle α such that a line going through P at angle α is in the distance d from the center of the image. The distance is then rounded to the nearest point in the discrete Hough space. We use angles from 0 to π and negative distance, so all parallel lines have the same angle but different distance.

For the pseudocode, see Algorithm 1. See figures 4.2 and 4.3 for an example of an image and its Hough transform.

O'Gorman and Clowes (1976) improved the Hough transform by using local gradient to reduce the number of useless votes and computation time. The Hough transform was further generalized to detect arbitrary shapes by Ballard (1981).

1.4 RANSAC

The RANSAC (*RANdom Sample And Consensus*) algorithm was first introduced by Fischler and Bolles (1981). It is a stochastic method for estimation of the parameters of a model from data contaminated by large amount of outliers.

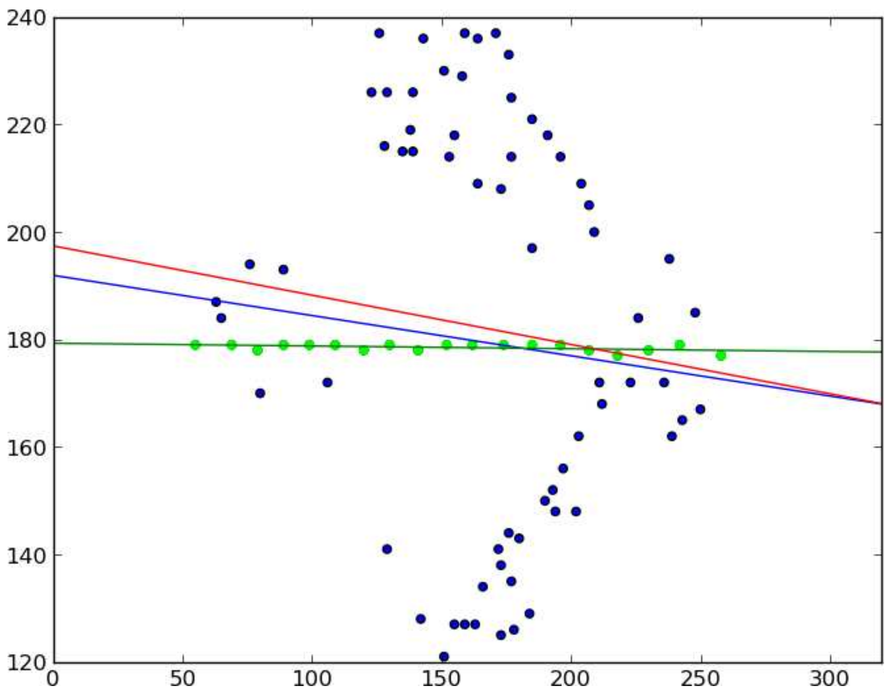

Figure 1.1: RANSAC estimate of a linear model. Dataset consists of all the dots, red line shows the model estimated by the least squares method, blue line shows the Theil-Sen estimate, green line shows the model estimated by RANSAC (with consensus set consisting of green dots).

We will illustrate the RANSAC process with an example of a linear model. Suppose we have some data generated from a linear model with Gaussian noise (e.g. green dots in Fig. 1.1). We would like to estimate the parameters of the model. In this case, we can just use the least squares method (the green line in Fig. 1.1) to estimate the parameters of the model.

Now we take the data generated from the same model, but this time with outliers (blue dots in Fig. 1.1). The outliers are distributed arbitrarily, therefore we cannot expect their deviations from the model to cancel out. As you can see, there can be quite a lot of outliers, but the line can still be distinguished. This time, the least squares (red line in Fig. 1.1) are quite off. Theil-Sen estimate

(Sen, 1968) is better, but still not good enough (blue line in Fig. 1.1). This is where RANSAC helps.

Algorithm 2 RANSAC algorithm

```
procedure RANSAC(data, distance, threshold, max_iter)
    $best \leftarrow \{\}$
    $i \leftarrow 0$
    while $|best| < threshold$ and $i < max\_iter$ do
        $consensus \leftarrow \text{RANDOM\_SAMPLE}(data)$
        $c \leftarrow 0$
        while $c < |consensus|$ do
            $c \leftarrow |consensus|$
            $model \leftarrow \text{ESTIMATE}(consensus)$
            $consensus \leftarrow \text{FILTER}(model, data, distance)$
        if $|best| < |consensus|$ then
            $best \leftarrow consensus$
        $i \leftarrow i + 1$
    return $best$
```

In each RANSAC iteration we randomly select exactly enough datapoints to determine the parameters of the model (2 datapoints in the two-dimensional linear case). Then we check all the datapoints against this model. We create a so called *consensus set*, consisting of datapoints that are closer than d to the model (d is a parameter). We assume that the consensus set consist of inliers, therefore we can use e.g. least squares method to create a new estimate of the parameters based on the consensus set. We create a new consensus set for the new model. We continue with re-estimating the model (based on the consensus set) and recomputing the consensus set (based on the model) until the consensus set stops growing. For common models and estimators, this part can be regarded as an instance of EM algorithm.

We repeat the whole cycle of randomly sampling the data and improving the model, until we find a suitable solution or reach a predetermined number of iterations. In Figure 1.1 we end with consensus set consisting of green dots and the corresponding model is visualized by the green line.

If we have an apriori estimate of the ratio of outlier and inliers, Fischler and Bolles (1981) give an estimate on number of iterations needed to achieve selected probability of finding the correct consensus set.

There are many variations to the RANSAC algorithm. One important modification is MSAC (Torr and Zisserman, 1998), where the inliers are scored by their fitness to the model and outliers are given constant weight. This does not present any computational overhead, but it can lead to better results. Another modification is MLESAC (Torr and Zisserman, 2000), where the consensus set is scored by its likelihood, instead of its cardinality.

For more information on RANSAC we recommend an excellent guide written by Zuliani (2014).

2. Analysis of the problem

In this chapter we analyze various potential solutions to the problem of optical position recognition in Go. In Section 2.1 we provide an overview of the problem and discuss the assumptions we make about the input. In sections 2.2 through 2.5 we examine individual stages of the game position recognition.

2.1 Overview

There are many features that we can use to find the board in the image. Human brains probably use many of them simultaneously. We could for example cluster the pixels of the image by their distance in color space. This would work in many cases, except for a fairly common situation of wooden board on a wooden table. The most reliable system would combine all of these features in order to work in any conceivable conditions. Unfortunately, because we are working with image data, the computations for such a system would take intolerable amount of resources. We need to carefully select what features we use in order to balance reliability with speed, if we want to build a system that is effectively usable.

This differs from general pattern recognition tasks in that we can exploit the geometric properties of the grid. In following paragraphs, we present various possibilities to do so.

2.2 Finding the grid

First step of the position recognition is locating the grid. We could try to locate the board and restrict the search for the grid to the appropriate area of the image, but this not necessary.

To simplify the analysis, we assume that there are no stones on the board. We deal with stones in the last paragraph of this section.

Detecting the edges To lower the amount of noise in subsequent stages of recognition, we might detect edges in the image and use them instead of the original image. We assume that the grid consists of dark lines on light background, therefore we can use a simple filter to detect edges. The filter approximates the Laplacian of the image, by computing a convolution sum with the following mask (so called *Laplace operator*):

$$L = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

There are many other methods for edge detection, see (Sonka et al., 2014) for a survey.

Finding the lines We find the edges. Then we use the Hough transform. Peaks in the Hough space correspond to strong lines in the image. Now we need to find the lines that belong to the grid.

Points representing lines that are parallel in the scene lie almost on a line in Hough space. They actually lie on a sinusoid, as lines that are parallel in the scene intersect at a vanishing point in the image plane and each point in the image is projected on a sinusoid in Hough space. But if we assume that we see the board under reasonable angle, then the vanishing point is far away from the center of the image, therefore the sinusoid formed by the vanishing point in the Hough space has a large amplitude (x or y are large in $d = x \cdot sin(\theta) + y \cdot cos(\theta)$). We are interested only in values relatively close to zero, because lines corresponding to larger values are out of the image. And the part of the sinusoid that is near the zero can be safely approximated by linear function.

Therefore we can easily find the two sets of lines that are parallel in the scene and form the grid, as they lie on two lines in Hough space. Some lines might be missing and we will usually find some extra ones (e.g. edges of the board).

Perspective We can use properties of perspective to try to find the grid analytically. If we assume two-point projection, we can find vanishing points for both sets of lines. They determine the horizon line. Intersections of grid-lines with any line parallel to the horizon are evenly spaced on the horizon-parallel line. The Fourier transform can be used to find the strongest frequency, which corresponds to the spacing of the grid-lines. Once we have the spacing, its easy to select a grid that maximizes the likelihood of observed lines.

There are also images, that are closer to one point perspective. Those can be analyzed similarly. For images with no point perspective (camera parallel to the board) the analysis is even simpler.

Finding the grid by minimization The problem of locating the grid in the image can be modeled as a minimization problem. To do so, we need to choose a method for generating candidates for the grid and a function to minimize.

Selecting the parameters is important part of the minimization formulation. The more information we can get from the analysis of the image (e.g. angles of the lines from the Hough transform), the less parameters we need. The less parameters we use, the faster the computation. On the other hand, if the information used to restrict the search space was inaccurate or even wrong, it prevents us from reaching the correct solution.

If we parametrize on positions of the corners, we have eighth parameters. If we consider the position of the camera relative to the board, we have six parameters. We can estimate the angles of lines from the Hough transform and take only their distance from the center of the image as parameters (then we have four).

Next we need to choose a cost function. The cost function can compare the candidate to the edges detected in the image, to the lines detected from the edges or their intersections. The first approach seems to be the most reliable, but it is also the most computationally demanding.

In any configuration, this is quite costly to compute. The problem is in its essence non-convex with many local minima (e.g. the position with grid shifted by one line). Therefore we cannot use gradient descent and must resort to stochastic optimization method such as Particle Swarm Optimization (Kennedy and Eberhart, 1995) or Cuckoo Search (Yang and Deb, 2009).

Selecting the lines We can try to build the grid from the lines that we found by the Hough transform. We must somehow deal with some extra lines and some grid-lines missing. Exploring all possible combinations is computationally expensive. But we can employ a RANSAC-inspired approach: select two lines in each direction at random, try various configurations of the grid (each of selected lines being interpreted as first, second, third, . . . , line of the grid), repeat this a few times, pick the best candidate. The score of the candidates can be any of the functions mentioned in previous paragraph.

Finding the center Another method is based on finding the center and diagonals of the grid first. Once we have the center and angles from the Hough transform, the grid is determined by just two corners. We can further narrow this down by finding the diagonals of the grid first. Because diagonals go through the largest number of intersections (except for the lines that we already have), they can be found with RANSAC.

Polygons Instead of finding lines in the image, we can try to detect polygons. We assume that most of the polygons in the image will be on the board and they will all be roughly the same size in the perspective projection. Therefore we can find the median distance and discard all polygons that deviate from it too much. Then we take the convex hull of the remaining polygons and we should end up with the grid.

Corners and intersections There are extensions to the Hough transform for finding the corners and intersections of lines (Barrett and Petersen, 2001). This could be used to find the intersections. The result would not contain false intersections, that we get when simply computing intersections of lines found by the Hough transform.

Stones on the board Detecting lines with Hough transform works even with stones on the board. With too many stones on the board, the lines get obscured and harder to detect. Rules of Go allow the board to be almost completely filled with stones. In that case we would need to detect stones instead of lines. Hough transform for circles could be used for that. But the board is highly unlikely to get completely covered in a real game. In Figure 2.1 we see two finished games with lines detected by the Hough transform. Some lines are missing, but there are clearly enough lines to find the grid.

2.3 Finding the stones

When detecting stones, we can ignore the perspective transformation. Under any reasonable angle, Go stones are projected as roughly circular shape, either black or white.

Based on color If we know the position of the grid, the simplest approach is to detect stones by their color. We can quantize the colors — select the nearest (black, white or brown) for each pixel — and then pick the most frequent one. Or

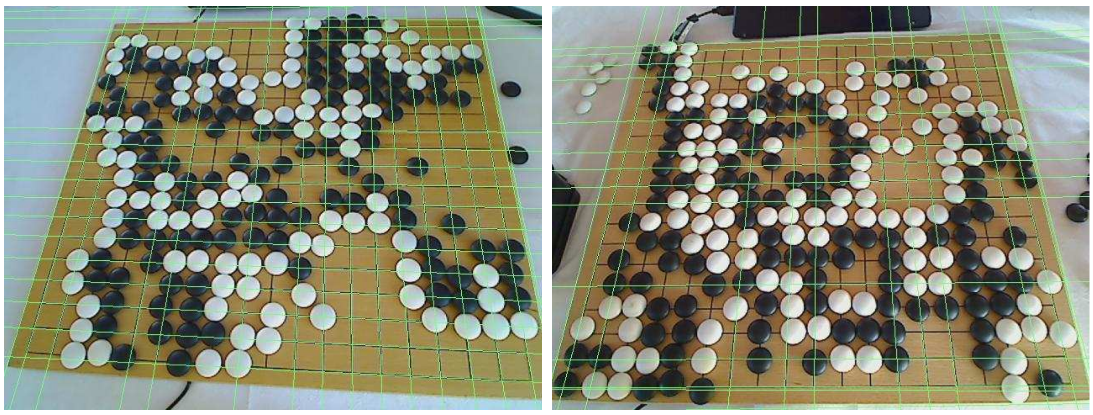

Figure 2.1: Finished games with lines found by Hough transform.

we can take the area around an intersection and compute average or median color. Because of variable lighting conditions, it is not possible to set hard boundaries to classify the colors. Instead, we need to cluster them. K-means clustering (Hartigan, 1975) is sufficient for this, although other variants of EM clustering might be more reliable, because the clusters differ in size significantly (at least in the beginning of the game).

Based on shape We can use shapes to detect if there is a stone on the intersection. Deciding if the stone is black or white is then straightforward. Hough transform can be used to detect circles. If we have the position of the grid, we can run the transform only around intersections. Complementary approach is to detect empty intersections (dark crosses on light background). Both methods can be combined.

Machine learning Another possibility would be to cut a square around each intersection from the image and use machine learning methods to train a classifier. This would work best if it was trained for a particular setup (camera, board, lighting conditions, etc.). It would require a large and very diverse training set to make this sufficiently general.

2.4 Sequences of images

In the game of Go, exactly one stone is added on the board in every move (although multiple stones can be removed). If we had exactly one image of the board for every move and a perfect program for analyzing a single image, the task of finding the one added stone would be trivial. The stones to remove would then be determined by the rules of Go.

In practice, we do not have a perfect program for single images. Some images will be interpreted incorrectly. We need to consider the relation between images to deal with this. We can regard the images as observations of a hidden Markov model (HMM), as suggested by Scher (2006) and later developed by Scher, Crabb, and Davis (2008).

If we regard the state of the game as a HMM, and the images as observations, we can set the parameters of the model in such a way that it filters out the errors in image analysis. For short games or smaller boards, the Viterbi algorithm (Viterbi, 1967) can be used to find the most probable order of the moves given the observations. For complete games on a 19×19 board, the state space is too large for an exhaustive search and we need to heuristicaly prune it.

2.5 Video

To analyze a video recording, we can just treat it as a sequence of images. It would be beneficial to detect and discard frames containing object (such as player's hand) moving over the board. This can be easily achieved by computing the difference between consecutive images.

If we end up discarding too much frames (e.g. in a recording of a blitz game, when players are constantly moving their hands over the board), we can use optical flow estimation (Sun et al., 2010) to find areas of strong movement. We then restrict our analysis to the static regions of each frame. Partial observations are sufficient for HMM estimation.

There is also a question of analyzing a live video stream (e.g. for broadcasting an important tournament game over the internet). The software built for this task needs to be very fast as well as reliable. The precision of such a system would be reduced by the fact that it would only know about past observations and could not alter its decisions in situations such as detecting a conflict with the rules and deciding that one of the past observations was flawed.

3. Related works

In this chapter we provides an overview of relevant academic work and currently available software.

3.1 Academic work

Optical recognition of Go position is an uncommon topic in academic writing. Only recently there has been a few papers published, the oldest we found dating from 2004.

None of the works mentioned below has publicly available working implementation.

Srisuphab et al. (2012) describes AutoGoRecorder, an application for live recording of Go games and record database management. It finds the board at the beginning of the recording and assumes the position does not change. The camera must be positioned directly above the board. The average color in HSV color space measured in circular area around intersection is used to detect stones. Movement detection is used to select the frames in which the board should be checked for changes. There is no evaluation of the performance and the application itself is not available.

Seewald (2010) presents a system based on machine learning methods. The corpus of images on which the system is trained and evaluated was created with a custom 8×8 board. There is no reason to assume that this approach can work for other users, unless everyone creates their own dataset and trains their own classifier specifically for their board and camera.

Scher (2006) proposes a method to improve a weak single-image classifier by taking the whole sequence of images into account. The images are treated as observations of states of a Hidden Markov Model. The Viterbi algorithm is used to find the most probable sequence of states. This approach can be used to improve any system that works on single images. This method can be further developed by using the A* algorithm to search the space of possible move sequences (Scher et al., 2008).

Kang et al. (2005) and Yanai and Hayashiyama (2006) both propose algorithms to extract game records from Go TV shows. Both works rely on the assumption that the video is recorded with a camera positioned directly above the Go board. This means that there is no perspective transformation and vertical lines on the board are also vertical in the image (and the same holds for horizontal lines). This is a standard setup in TV broadcast of Go events, but its use anywhere else is very limited. It is fairly easy to analyze images under such strict assumptions and both works deal mainly with methods of selecting correct frames from the TV broadcast.

Hirsimäki (2005) describes a MATLAB prototype of a system for recognizing Go positions in single images. It is based on Hough transform. The two sets of parallel lines are found by projecting the Hough space onto the angle axis and locating two peaks. Small sub-grid is selected in the estimated center of the grid. The sub-grid is then grown until it fills the whole grid. The stones are classified by average luminosity around intersection. The output of the prototype consists of an image with marked grid and stones, no abstract representation of the position is given. The performance was evaluated on six images that can be downloaded with the prototype.

The C++ implementation is also available[1], but there was no update since 2008. We were not able to compile it, but is clearly intended to only output image with marked situation, just as the prototype.

Shiba and Mori (2004) suggest to use a genetic algorithm to find the contour of the board in the image. They consider this being the first step towards extracting the game position from images, but do not deal with subsequent steps (grid and stone location) at all.

3.2 Available software

There is a lot of software that attempts to automate Go game recording. Unfortunately, most of it is not complete and it is unmaintained and poorly documented. In this section, we survey available programs, the principles they work on and their current state of development.

Lot of the applications require to specify grid position manually and therefore we cannot compare their performance with our work, which solves a harder problem.

Rocamgo[2] only works with video. It finds the corners of the grid (and assumes the camera does not move). Then it applies the reverse perspective transform and finds the stones in the normalized image. Unfortunately, the stone-finding part seems to be very unreliable. In a video with 30 first moves of a game, it only found 22 moves and their order was entirely wrong. The documentation is in Spanish, therefore we do not know what principles it works on, apart from what is obvious from the images displayed when the program is running.

MHImage/BenjaminsBox Go Image Recognition[3] can analyze video, but requires user to manually select proper frames, indicate the grid and manually calibrate stone recognition.

image2sgf[4] requires user to supply coordinates of the corners of the grid. It quantizes colors to classify the position. It was last updated in 2009.

[1] http://koti.kapsi.fi/thirsima/gocam/

[2] http://github.com/Virako/Rocamgo

[3] http://www.public.iastate.edu/~bholland/project_archive_content/GoImageRecognition.html

[4] http://www.inference.phy.cam.ac.uk/cjb/image2sgf.html

ludflu/kifu[5] finds all closed polygons with 4 sides, it keeps those with area around median and computes the convex hull. Implementation is not finished, last update was made in 2012. See Figure 3.1 for result of the computation on one image from our dataset.

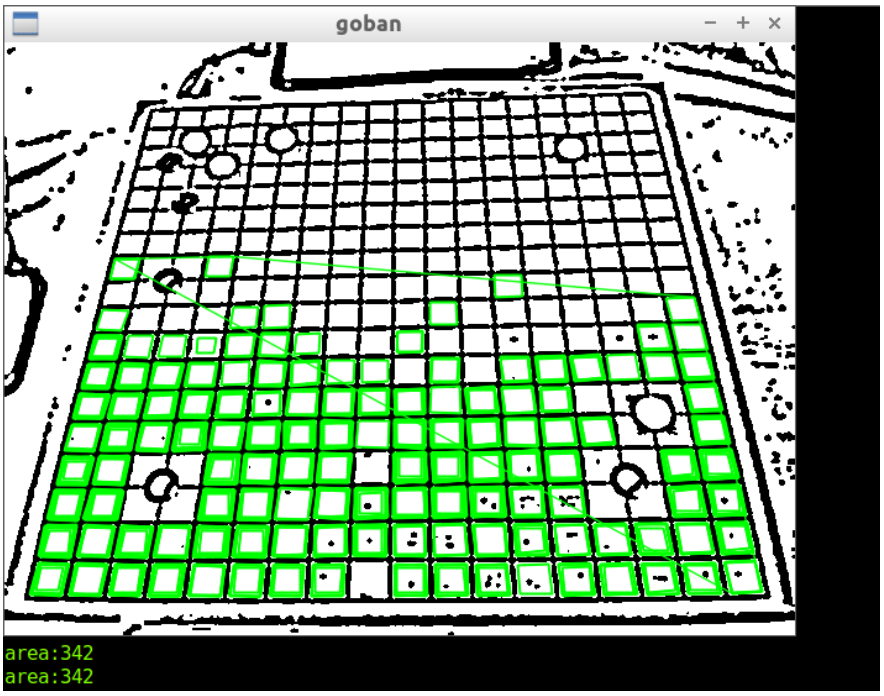

Figure 3.1: Screenshot of ludflu/kifu.

Kifu[6] starts by finding a big quadrilateral shape. It assumes that is the board, cuts it from the image and uses Hough transform to find the lines. The program relies on the assumption of static board and camera. The homepage claims that it works on video, but the documentation[7] does not describe how to choose relevant frames. It seems to be no longer maintained and it can no longer be downloaded.

Go Tracer[8] is a web application. It asks the user to manually indicate the position of the grid. After that, it clusters the colors around intersections and produces a diagram. The color classification seems to be quite unstable.

Kifu-Snap[9] is an Android application. The processing is done on the phone.

Go Scoring Camera[10] is an Android application for automatic counting of the score. Processing is done on a remote server. Both this and Kifu-Snap are paid and neither one is extensively documented.

[5] http://github.com/ludflu/kifu
[6] http://www.sourceforge.net/projects/kifu
[7] http://wiki.elphel.com/index.php?title=Kifu:_Go_game_record_(kifu)_generator
[8] http://go-tracer.appspot.com/
[9] http://remi.coulom.free.fr/kifu-snap/
[10] https://play.google.com/store/apps/details?id=com.jimrandomh.goscoringcamera

Compogo[11] was a SGF file editor for Windows, that was able to analyze series of images, given the position of the grid. The development was stopped in 2003.

Go Game Recorder[12] webpage has a video demonstration, but no actual software or source code. It was last updated in August 2013.

Go Watcher[13] webpage also has a video demonstration, but no actual software.

[11] http://senseis.xmp.net/?CompoGo
[12] https://gogamerecorder.wordpress.com
[13] http://www.gowatcher.com/

4. Game position recognition

In this chapter we explain our solution and give an example of image analysis. The most significant difference from previous works is our use of RANSAC in the analysis of Hough transform and in subsequent proccess of finding the grid. Another significant improvement is the use of clustering in the stone finding phrase. We also propose a new method for analysing video.

4.1 Finding the grid

In Figure 4.1 on page 19 we see the input image. It was taken by a webcamera connected to the computer that can be seen in the top of the image. The computer was running a Go clock and taking a picture each time a player pressed the key after his move. This method of recording lets the players to focus on the game (as tournament players are already used to playing with clock).

First we convert the image to gray-scale by computing luminance for each point. To reduce the amount of noise in the subsequent Hough transform, we apply edge detection. We assume the lines are dark on light background, therefore we use a simple filter with Laplace operator, followed by a high-pass filter (Fig. 4.2).

Next we take the Hough transform (Fig. 4.3, p. 20). Bright points in the Hough transform represent strong lines in the image. The Hough transform is again followed by a high-pass filter. If the lines are not strong enough, we can use a filter similar to the one we used for edge detection (only with flipped signs) between the Hough transform and high-pass filter. This results in small groups of non-zero pixels, so we find connected components in the image and replace each of them with its center. This way, we get one point in Hough space for each line in the image.

The grid is formed by two sets of parallel lines, so we are searching for two lines in the Hough space. We use RANSAC to find them (Fig. 4.4). We use standard RANSAC to find the first line, we delete the points that lie on it and then we use RANSAC again to find the second line. It would be possible to formulate a model with two lines, but it would need many more iterations to get the right solution.

In Figure 4.5 on page 21 we see the selected lines over the original image.

Now we find the candidates for the center of the grid. We make use of the fact that we have most of the lines, therefore most of their intersections. If we exclude the lines that we already have, the lines that contain most points are the diagonals of the grid. We use RANSAC to find them (Fig. 4.6).

Once we have candidates for the diagonals, we can generate candidates for the grid. We take an intersection on a diagonal as a corner of the grid. Because we have the lines and the other diagonal, we can compute position of remaining corners and therefore the whole grid. We repeat this for all the intersections on all the candidates for diagonals and select the best grid (Fig. 4.7, p. 22). The grid candidates are scored in the same way as models in MSAC: for each of the intersections found in the previous step we count its distance to the nearest line of the grid if it is smaller than δ and δ otherwise. We found that $\delta = 2$ works well for our application.

4.2 Finding the stones

For each intersection in the grid, we take average color in a 5×5 pixels square around it. In Figure 4.8 we see the average colors distributed in a color space, where the horizontal axis represent luminance and the vertical axis represent saturation.

We cluster the colors into three clusters using k-means clustering (Fig. 4.9, p. 23). The leftmost cluster corresponds to black stones, the rightmost to white stones. The middle cluster corresponds to empty intersections.

Once we have assigned stones to intersections, we can print the result as in Figure 4.10.

4.3 Video

In this section, we discuss our proposal for the video-analysis extention of the work described above. This exceeds the range of the implementation part of this thesis and its realization is left for future works.

Trying to select precisely the frames in which we can observe the state of the board between moves and then analizing them without mistakes seems like a very unreliable approach. Therefore we propose to follow Scher et al. (2008) and use a HMM to model the game. As the state-space is quite large (up to 361^M where M is number of moves in the game), it is not feasible to compute the best path with Viterbi algorithm. Scher et al. (2008) find a good path with modified A* algorithm. We propose to search the state space with Monte-Carlo Tree Search (MCTS, Coulom (2006)) instead. MCTS was already used in computer Go programs with great success. There are open-source implementations of MCTS that already contain Go rules, modifying one to count with probabilities instead of number of wins should be straightforward.

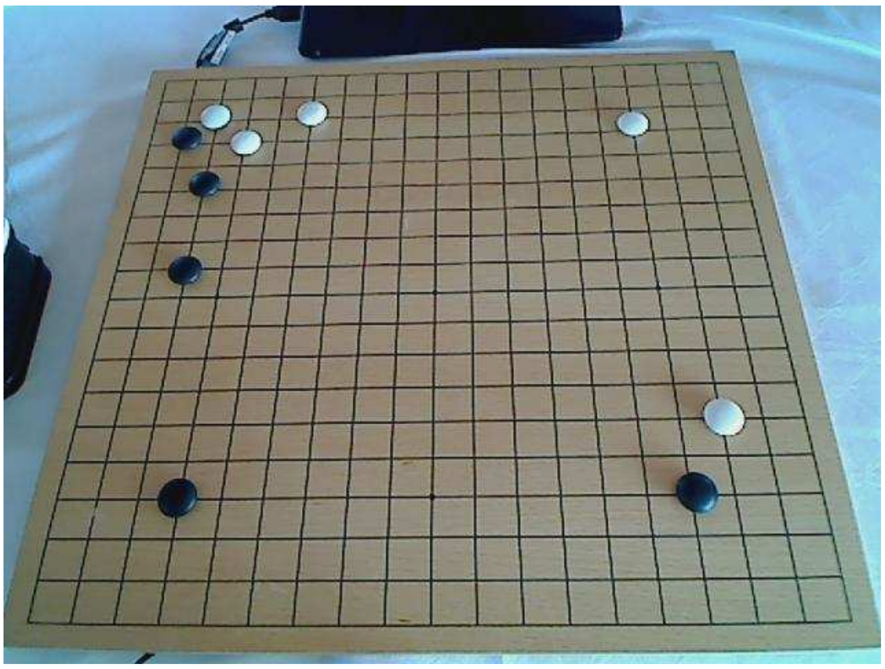
Figure 4.1: Original image.

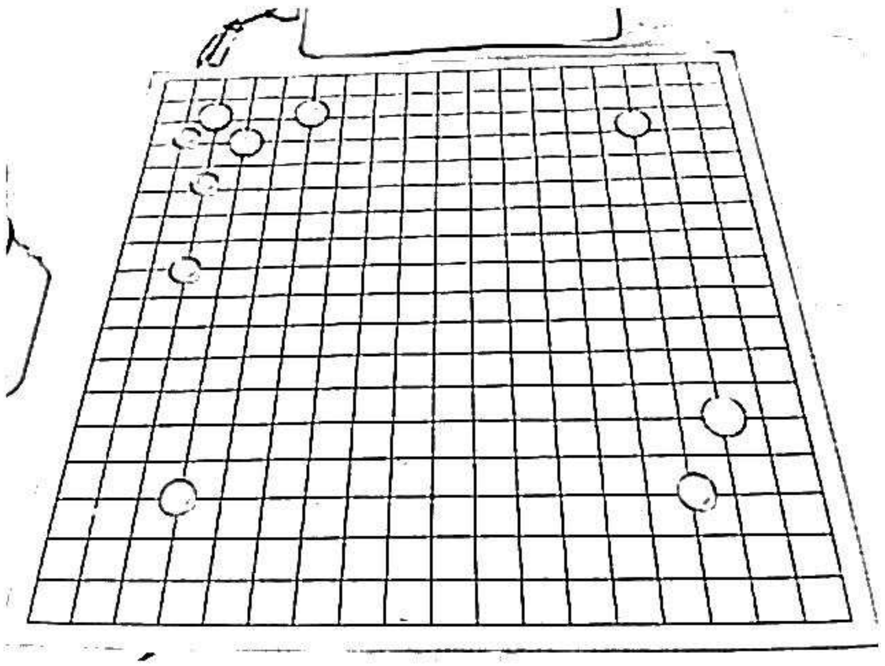
Figure 4.2: Edge detection (colors inverted).

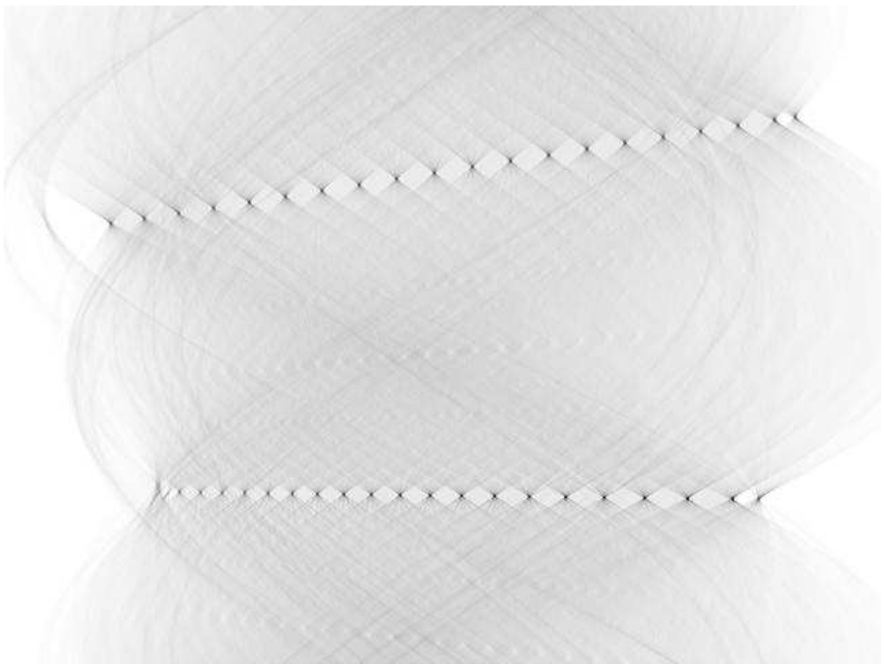
Figure 4.3: Hough transform (colors inverted).

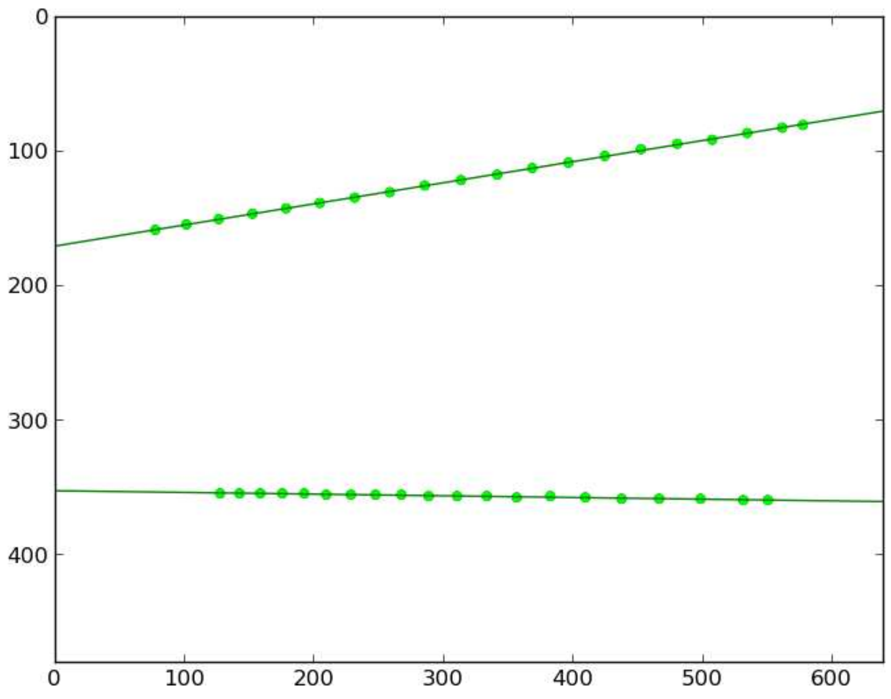

Figure 4.4: RANSAC estimate of two lines.

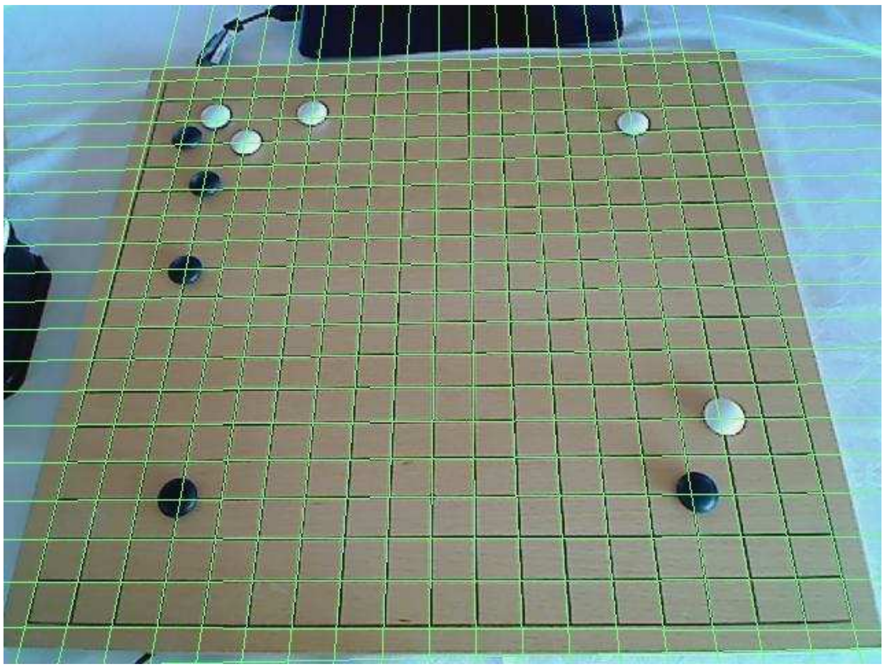
Figure 4.5: Lines over the original image.

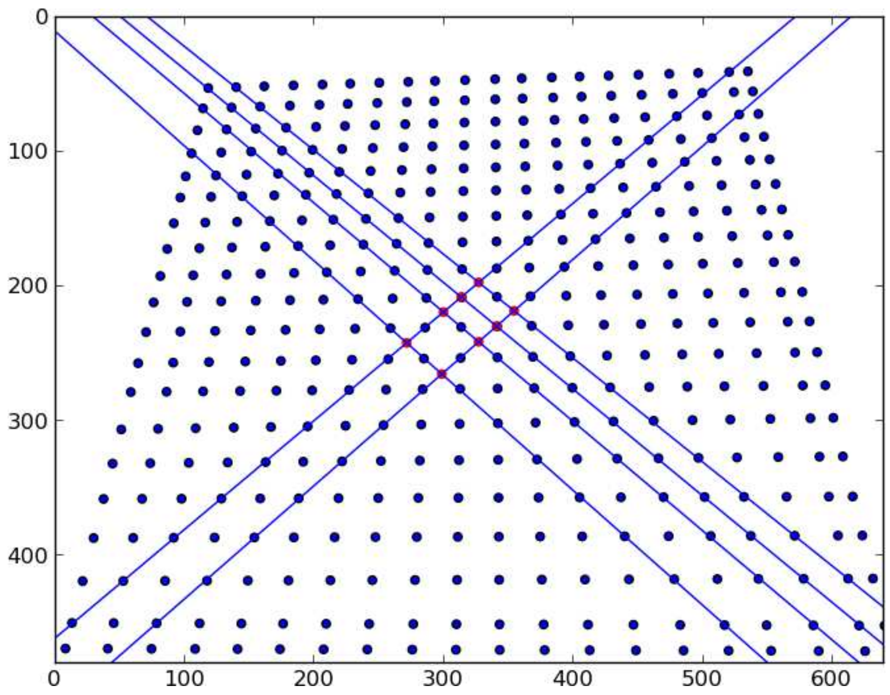

Figure 4.6: Finding the center.

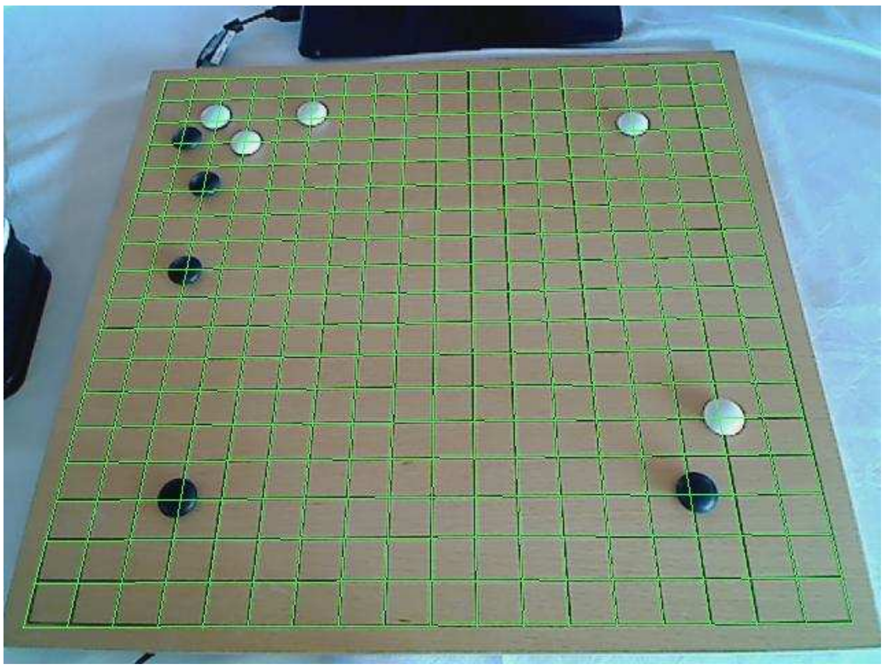

Figure 4.7: The grid.

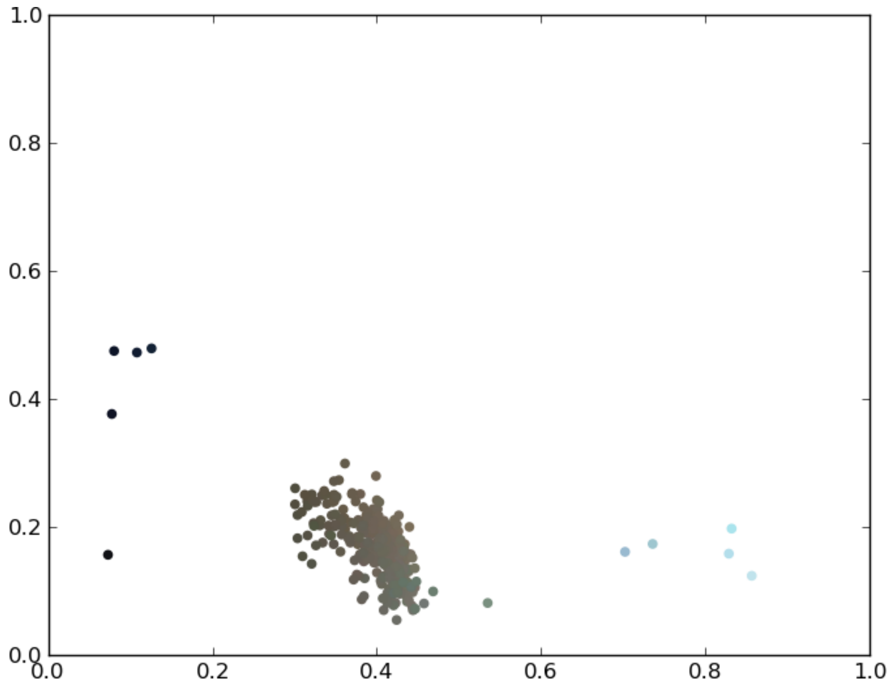

Figure 4.8: Distribution of the color around intersections.

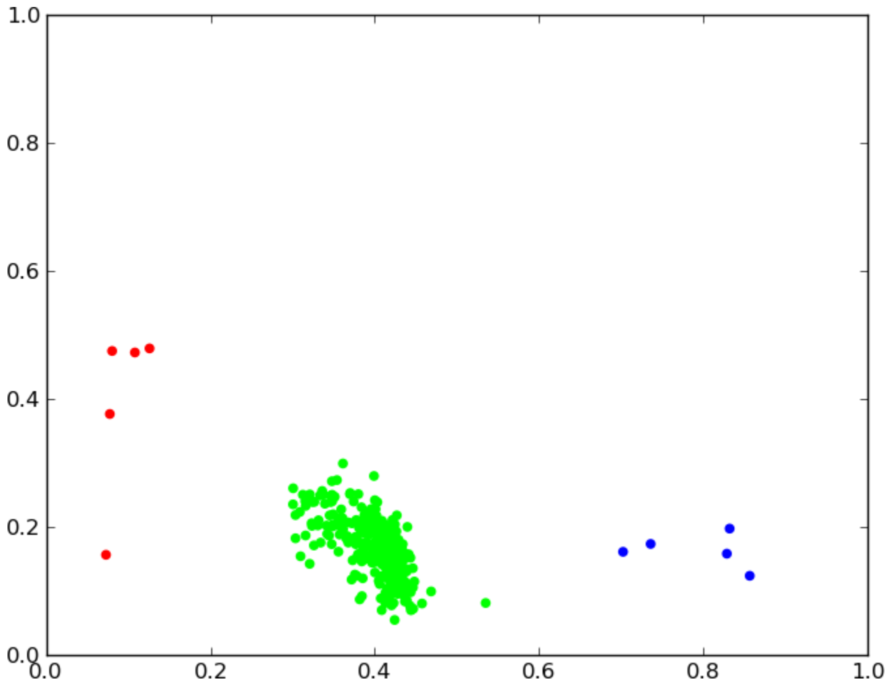

Figure 4.9: Clustering.

```
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . W . . W . . . . . . . . . . . . .
. B . W . . . . . . . . . . . W . . .
. . . . . . . . . . . . . . . . . . .
. . B . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . B . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . W . .
. . . . . . . . . . . . . . . . . . .
. . . B . . . . . . . . . . . B . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
```

Figure 4.10: The result.

5. Implementation

In this chapter we describe Imago, our application for optical recognition of Go game positions. Section 5.1 offers general description of the application. In Section 5.2 we discuss high-level technical choices we made. Detailed developer documentation is distributed with Imago and can be found on the attached CD (see page 33).

5.1 Imago

Imago is a program for automatic processing of Go images. It takes an image (or a set of images), finds the board-grid and stones and produces an abstract representation of the game situation (or a game record).

It supports JPEG, BMP, TIFF (and other) image formats on input. It is capable of output to ASCII and SGF format. As the process should be fully automatic, the program is operated from command line. There is, however, a GUI for manual grid location in case the automatic one should fail.

The program is written mainly in Python, with performance-critical parts written in C. It is distributed under slightly modified *BSD 3-Clause License*[1].

The source code is available on GitHub[2].

5.2 Technical documentation

Libraries and other dependencies

Most of the program is written in Python 2, so the Python 2 interpreter is needed to run it. Python development files and C compiler are needed to compile our `pcf` module.

Beside Python's standard library, we use PIL[3] (or a compatible library, like Pillow[4]) for image manipulation and numpy[5] (Van Der Walt et al., 2011) for array computations and linear algebra. These libraries should be sufficient for basic functionality of Imago. For showing intermediate computations and plots, matplotlib[6] is needed. Graphical user interfaces are implemented with pygame[7]. For capturing images with a camera we use OpenCV[8] (Bradski, 2000).

Conventions

image coordinates The coordinates of the image start in the top left corner with $[0, 0]$. Fist coordinate is horizontal, second is vertical.

[1]`http://opensource.org/licenses/BSD-3-Clause`
[2]`http://github.com/tomasmcz/imago`
[3]`http://www.pythonware.com/products/pil/`
[4]`http://python-pillow.github.io/`
[5]`http://www.numpy.org/`
[6]`http://matplotlib.org/`
[7]`http://www.pygame.org/`
[8]`http://opencv.org/`

lines We use multiple representations for lines, depending on the context. Lines represented by two points are easy to draw into image. For geometric computations (distance from a point, intersection of two lines, etc.) we represent lines by coefficients of the linear equation in general form ($ax + by + c = 0$). Lines we get from the Hough transform are represented by pairs (θ, d), where θ is angle between 0 and π and d is the distance from the centre of the image. The distance can be negative, so that all parallel lines have the same θ and differ only in d.

colors There are two different representations of colors, one with discrete values from 0 to 255 for each channel, the other with floating point number between 0 and 1 for each channel. We mostly use the former, but some libraries (matplotlib, colorsys from the Python standard library) use the latter, therefore we need to convert between them sometimes. For gray-scale images we use discrete values between 0 and 255.

Most important modules

We give a high-level overview of the program architecture here. The detailed documentation is distributed with the program and can be found on the attached CD.

imago.py This is the main module of the program. It contains the user interface.

linef.py This module finds the lines in the image.

gridf.py This module finds the grid, given the lines.

intrsc.py This module finds the stones on the grid.

hough.py This module implements the Hough transform.

ransac.py This module implements the RANSAC algorithm.

k_means.py This module implements k-means clustering.

pcf.c This is a Python module written in C, that contains functions that are critical for fast performance (e.g. the image processing part of Hough transform that is called from `hough.py`).

Input and output formats

Imago loads images via PIL. It can open any image format, that is supported by PIL (JPEG, BMP, TIFF, . . .).

Output for single images is a simple textual board representation with '`.`' for empty intersection, '`B`' and '`W`' for black and white stones. See Figure 4.10 on page 23 for an example. For game records, resulting from analysis of image sequences, we use SGF file format. Single game positions can also be written with SGF setup syntax.

SGF file format SGF[9] (Smart Game Format) is the de facto standard file format for Go game records. Each move is recorded by its color (`B` or `W`), followed by coordinates in square brackets. The axes are labeled with letters from `a` to `s`.

There are a number of software packages for reviewing and editing Go SGF files (see article on *Sensei's Library*[10]).

```
(;
SZ[19] HA[0] ST[0]
PB[Shusaku] PW[Gennan Inseki]
KM[0.0] RE[B+2]
BR[4d] WR[8d]
C[Gennan Inseki(white) VS Shusaku(black)]
;B[qd];W[dc];B[pq];W[oc];B[cp];W[cf];B[ep];W[qo];B[pe];W[np]
;B[po];W[pp];B[op];W[qp];B[oq];W[oo];B[pn];W[qq];B[nq];W[on]
;B[pm];W[om];B[pl];W[mp];B[mq];W[ol];B[pk];W[lq];B[lr];W[kr]
;B[lp];W[kq];B[qr];W[rr];B[rs];W[mr];B[nr];W[pr];B[ps];W[qs]
;B[no];W[mo];B[qr];W[rm];B[rl];W[qs];B[lo];W[mn];B[qr];W[qm]
)
```

Figure 5.1: Example of a SGF file — first 50 moves of a game.

Requirements for input images

We expect that:

- the board is standard 19 × 19 Go board, it has dark lines on light background,
- the board is completely visible — it does not exceed the image,
- no part of the board is obscured by player's hands or other object getting between the board and the camera,
- the image is reasonably sharp and the perspective is not exceedingly distorted,
- the resolution of the image is at least 640 × 480 pixels, and
- the board takes up most of the image — at least half of the shorter dimension of the image.

We do not have have any special requirements on camera angle and lighting conditions. If the image meets the criteria above and the position can be recognized by a human without difficulties, then the computer should be able to recognize it as well. The program might work even if the image violates some of the above mentioned conditions, but it is not guaranteed to.

[9] http://www.red-bean.com/sgf/
[10] http://senseis.xmp.net/?SGFEditor

6. Evaluation

In this section we evaluate the performance of Imago. We have not found any other work that we can meaningfully compare our results with (see Chapter 3 for details). Therefore we build our own dataset, that we describe in Section 6.1. Results of our experiments on the dataset are given in Section 6.2.

6.1 Dataset

There is no standard dataset for measuring performance in optical position recognition in Go. We collected a set of images and manually recorded their game positions. We hope that this dataset can be used as a basis of comparison for any future works in this area, despite its small deficiencies: Even though we tried to make the dataset diverse, we only had two different cameras and two different Go boards at our disposal. Because Imago was tested on this dataset during development, it would have to be expanded or replaced by an independent dataset to make the comparison with Imago fair.

We also recorded two complete games, with an image taken after each move. These can be used to experiment with extracting game records from image sequences.

6.2 Experimental results

Precision

The dataset consists of 55 images. We run the test 3 times, because parts of the algorithm are stochastic and results may vary.

	mistakes				**no mistakes**
	incorrect grid	**one stone**	**two stones**	**three stones**	
1	0 (0.0%)	3 (5.5%)	5 (9.1%)	5 (9.1%)	42 (76.4%)
2	0 (0.0%)	3 (5.5%)	5 (9.1%)	5 (9.1%)	42 (76.4%)
3	0 (0.0%)	3 (5.5%)	5 (9.1%)	5 (9.1%)	42 (76.4%)

Table 6.1: Test results on our dataset.

As we see in Table 6.1, Imago gives consistent results. The position of the grid was always determined correctly. There are a few images in which Imago misclassified up to three stones, but never more than that. Most of the images (76 %) were classified with no mistakes. If we employed a measure counting with the total number of intersections, Imago would have 99.9% accuracy on this dataset.

Detailed reports of test results can be found on the attached CD.

We also evaluated a version of Imago based on Cuckoo Search. With high number of iterations, the number of correctly determined grid positions was also nearly 100 %, but the analysis took on average 240 seconds for an image.

Time consumption

On Intel Core2 Quad Q6600 (2.40GHz) CPU it takes 28 seconds on average to analyze an image from our dataset. Most of the time is spent finding the diagonals with our variation of RANSAC algorithm. To reduce the time consumed by this part of the analysis, we can reduce the number of RANSAC iterations. In Table 6.2 we see how this affects the precision of our system. The time is reduced linearly with the decreasing number of iterations. Error rate increases significantly under 200 iterations. In the other direction, error rate decreases slowly and we need at least 800 iterations to consistenly achieve 0 mistakes on our dataset. This behavior was anticipated, because 200 iterations is the expected amount needed to find the correct diagonal line.

iterations	800	500	200	150	100
grids not found	0 (0.0%)	1 (1.8%)	2 (3.6%)	9 (16.4%)	14 (25.5%)
average time [s]	28	20	13	12	10

Table 6.2: The effect of number of RANSAC iterations on time and precision.

Another way to make our application faster is to rewrite our RANSAC implementation from Python to C.

Conclusion

We examined possibilities of optical position recognition in the game of Go. We developed a new method based on Hough transform and RANSAC. Based on our method, we implemented a state-of-the-art application for Go position recognition in static images. We released the application as free and open-source software. We also collected a dataset that can be used to measure performance of applications in Go position recognition.

We are convinced that with just a few adjustments and extensions, this system can be made so reliable and fast, that it could automatically record tournament games or even broadcast them over the internet without human supervision.

Future works

Most important future work is to extend this system to analyze video, as we describe in Section 4.3.

Finding of the grid can be made faster by reimplementing critical parts of the RANSAC algorithm in C.

The application can be made even more reliable by implementing stone detection based on shape (as we describe Sec. 2.3).

Principles described in this thesis can be used to implement systems for automatic recording of other games with geometrical board design, such as Hex or Shogi. For Gomoku and Renju our method will work with little or no modification.

Bibliography

Dana H Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern recognition*, 13(2):111–122, 1981.

William A Barrett and Kevin D Petersen. Houghing the hough: Peak collection for detection of corners, junctions and line intersections. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 2, pages II–302. IEEE, 2001.

Robert Charles Bell. *Board and table games from many civilizations*, volume 1. Courier Dover Publications, 1979.

G. Bradski. The opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

Richard O Duda and Peter E Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

John A Hartigan. Clustering algorithms. 1975.

T Hirsimäki. Gocam: Extracting go game positions from photographs. *Helsinki University of Technology, Helsinki, Finland, Tech. Rep*, 2005.

Paul VC Hough. Method and means for recognizing complex patterns, December 18 1962. US Patent 3,069,654.

Kaoru Iwamoto. *Go for beginners*. Ishi Press, 1972.

George H Joblove and Donald Greenberg. Color spaces for computer graphics. In *ACM siggraph computer graphics*, volume 12, pages 20–25. ACM, 1978.

Deuk Cheol Kang, Ho Joon Kim, and Kyeong Hoon Jung. Automatic extraction of game record from tv baduk program. In *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on*, volume 2, pages 1185–1188. IEEE, 2005.

J Kennedy and R Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

Frank O'Gorman and MB Clowes. Finding picture edges through collinearity of feature points. *Computers, IEEE Transactions on*, 100(4):449–456, 1976.

Steven Scher. Recording a game of go: Hidden markov model improves a weak classifier. `http://classes.soe.ucsc.edu/cmps290c/Winter06/proj/stevenreport.doc`, 2006. Accessed: 9. 7. 2014.

Steven Scher, Ryan Crabb, and James Davis. Making real games virtual: Tracking board game pieces. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

Alexander K Seewald. Automatic extraction of go game positions from images: A multi-strategical approach to constrained multi-object recognition. *Applied Artificial Intelligence*, 24(3):233–252, 2010.

Pranab Kumar Sen. Estimates of the regression coefficient based on kendall's tau. *Journal of the American Statistical Association*, 63(324):1379–1389, 1968.

Kojiro Shiba and Kunihiko Mori. Detection of go-board contour in real image using genetic algorithm. In *SICE 2004 Annual Conference*, volume 3, pages 2754–2759. IEEE, 2004.

Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

A Srisuphab, P. Silapachote, T. Chaivanichanan, W. Ratanapairojkul, and W. Porncharoensub. An application for the game of go: Automatic live go recording and searchable go database. In *TENCON 2012 - 2012 IEEE Region 10 Conference*, pages 1–6, Nov 2012.

Deqing Sun, Stefan Roth, and Michael J Black. Secrets of optical flow estimation and their principles. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2432–2439. IEEE, 2010.

Phil Torr and Andrew Zisserman. Robust computation and parametrization of multiple view relations. In *Computer Vision, 1998. Sixth International Conference on*, pages 727–732. IEEE, 1998.

Philip HS Torr and Andrew Zisserman. Mlesac: A new robust estimator with application to estimating image geometry. *Computer Vision and Image Understanding*, 78(1):138–156, 2000.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13 (2):260–269, 1967.

Keiji Yanai and Takehisa Hayashiyama. Automatic" go" record generation from a tv program. In *Multi-Media Modelling Conference Proceedings, 2006 12th International*, pages 4–pp. IEEE, 2006.

Xin-She Yang and Suash Deb. Cuckoo search via lévy flights. In *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214. IEEE, 2009.

Marco Zuliani. Ransac for dummies. http://vision.ece.ucsb.edu/~zuliani/Research/RANSAC/docs/RANSAC4Dummies.pdf, 2014. Accessed: 13. 7. 2014.

A. User documentation

Installation

Make sure you have all dependencies installed. That is PIL (or equivalent) and numpy for basic functionality, matplotlib, pygame and OpenCV for additional features. On a Debian-based system, you can just run `sudo apt-get install python python-dev python-imaging python-numpy python-matplotlib python-pygame python-opencv`. Most of the libraries can be also installed with PIP, except for pygame, which has to be downloaded from its website.

Once you have all the dependencies, you need to compile Imago C code. Run `make` in the `imago` directory. Imago should now be ready to run.

User guide

Command line interface

Using Imago is easy. Just run `./imago image.jpg` to extract the position from `image.jpg`. If you specify multiple images, Imago will find the grid in the first one and extract the game record, assuming that the board position does not change, and images are in the correct order with one image per move. With `./imago -S <files>` the game record is produced in SGF. Run `./imago -h` for help and the list of all options.

Graphical user interface

The goal of this work is to make Go recording as automatic as possible. As a result, we expect minimum of user interaction. Therefore we decided to design our graphical user interfaces in rather minimalist way.

Manual grid selection You can select the grid manually, by running `./imago -m <file> [more files]`.

Camera, Go timer There are two helper programs to facilitate taking of photographs during the game. The `./imago-camera` command, that takes images at the press of the key or automatically with selected interval. And `./imago-timer`, that displays Go game clock and takes an image with every clock-press. For detailed instructions on setting the time limits and video device attributes, run the command with `-h` option.

B. The attached CD

The content of the attached CD:

- source code of the implemented software (`imago`),
- developer documentation (`documentation`),
- a dataset consisting of images and text files with corresponding positions (`imago/test_data`),
- detailed test results (`imago/test_data`), and
- a PDF file with this thesis (`thesis.pdf`).

All the materials can also be found on the webpage of the Imago project[1].

[1] http://tomasm.cz/imago