Analyzing a Game's State with Computer Vision

Andrew Carpenter, Drew Gillie, Bobby Picciotti

Inspiration

With AlphaGo being in the news lately as an example of a groundbreaking development in the field of AI, we are reminded that board games present a great opportunity to push the field of computer science in in many different directions. One problem that complex systems like AlphaGo[7] and Chinook[1] (the super computer that solved Checkers) have with interacting with people in a natural way is that they represent their games in a far more abstract way than we do. We want to find a way to address this problem, and we think that one possible way to do this is through computer vision.

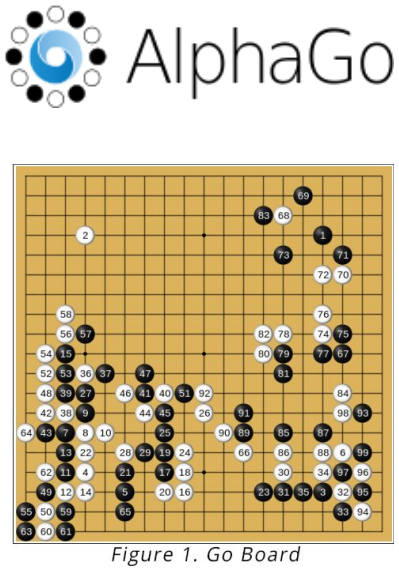

Figure 1. Go Board

These kinds of systems are also useful when testing the computer in real world competition. The test of AlphaGo against Lee Sedol makes for a good example. In the game a human player acted as an intermediary for AlphaGo. This allowed for a "real" human delay to move pieces before stopping the clock for AlphaGo's turn.

Purpose and Goals

The goal of our project is to be able to develop a system that can, given a picture of a board game and an indicator of the game being played, accurately interpret the state of the game represented in the photo. An example of a game suitable for a computer vision system like this is checkers. Its pieces and game board should be relatively easy to detect, and it has a simple ruleset that can be easily represented in code.

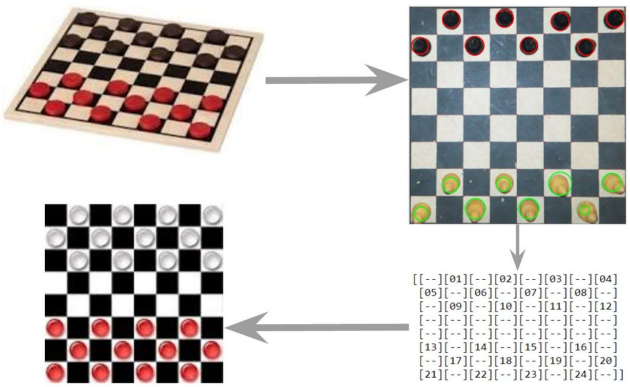

Figure 2: Conceptualized workflow diagram. Clockwise from top left:

Photo of game board, identified pieces on board, abstract representation of game, digital representation of game.

Though our main goal of this project is to simply be able to represent the board state, future work could involve AI to actually play the game. Checkers would make a good case study as a "solved" game; in that its outcome can be predicted from any position[1]. It might make for an interesting example of an entirely self-contained game playing robot: complete with vision to see the board, AI to play the game, and mechanical parts to move the pieces; but that would be beyond the scope of this project.

Algorithm and Pipeline

Needed Information

Originally, we decided that we would aim to determine the following thing:

- Location of board spaces
- Orientation of board
- Location of game pieces
- Color of each piece
- Game type

However, during the project itself we ended up scrapping doing other games and instead restricted ourselves to othello. This changes the problem of what we needed to determine somewhat, while leaving many of the key elements in place.

A list of what we now work to determine is

- Location of board spaces
- Location of game pieces
- Color of each piece

The type of game was removed obviously. We also removed the goal of determining board orientation. Othello is played in such a way that orientation of the board doesn't change how you play the game. So, while we do identify the corners, it wouldn't actually be possible to identify a true orientation to the board.

The Algorithm

Of course, the first step is to obtain an image for the program to run on. This is done manually (originally we had hoped to automate this, but it proved impossible). The program has a hardcoded filename for what file to use.

From here the image needs to actually be analyzed. Othello is played on a green board with white and black pieces. We take advantage of this sharp contrast of colors in our algorithm. The algorithm starts by filtering out all non-green colors and then inverting the image. The equation to do so is based on a matlab tutorial for green screens [8]. After finding the greenness of each pixel a filter is applied.

This process does an excellent job of identifying the board and leaving holes where the pieces are. Regionprops is used for the identification of the board within a bounding box. This does leave the algorithm subject to interference of green objects in the image outside the board. Such an object would cause the bounding box to not properly close around the board. However, if the box successfully bounds the board then the algorithm will be immune to interference outside of the board. This is useful as the board of an actual game would be expected to be clean, but stray objects could easily exist around the board.

With the board identified by the bounding box, the pieces are the next object that need identification. Regionprops is used again for identifying the pieces. Connected components is used for figuring out the centroid and size of each piece in the image. Only centroids within the bounding box are examined, preventing outliers outside the board. Connected components also provides the area of the component. So, that information is used to determine if the identified object is actually a circle.

To identify where a piece is actually located within the game (as opposed to just in the image) a grid will need to be created that reflects the game board. This is done by using the bounding box of the board. Othello is played on an eight by eight board. So, given that we know where the corners of the board are extrapolating to the grid is straightforward. Simply placing a line every seventh of the width of the board is quite effective, if simplistic. This method is not perfect, however, given that the centroid is in the center of the actual grid, some error does not throw off the result wildly.

Finally, the color of the pieces needs to be determined. To do this, a color channel needs to be chosen to do the majority of the processing of the images on. The channel chosen was the green channel because it is the brightest channel of images. The green channel also happens to line up fairly close with the V Lambda curve for photopic vision. This curve defines the way humans perceive lightness. [9] This will also be the sharpest channel and because cameras are made with twice the amount of green sensitive pixels than blue or red the obvious choice is the green channel.

The final image pipeline we formulated is quite robust, and can handle images taken with different cameras in various lighting conditions (provided the image is in the proper orientation and angle: straight above the board). Our final implementation can be evaluated by running the file `green_seg.m`, which results in a labeled image of the board and output of the board state on the console.

Developing the Algorithm

First process

The first process done to tackle this challenge was to take an image of a board, and an image of a board with pieces on it. The images could be subtracted to get either white pieces or black pieces separate from everything else in the image.

An app of the game Go, and Othello was played images were captured via screenshot. This was then piped into the first version of the algorithm to see where this technique would go.

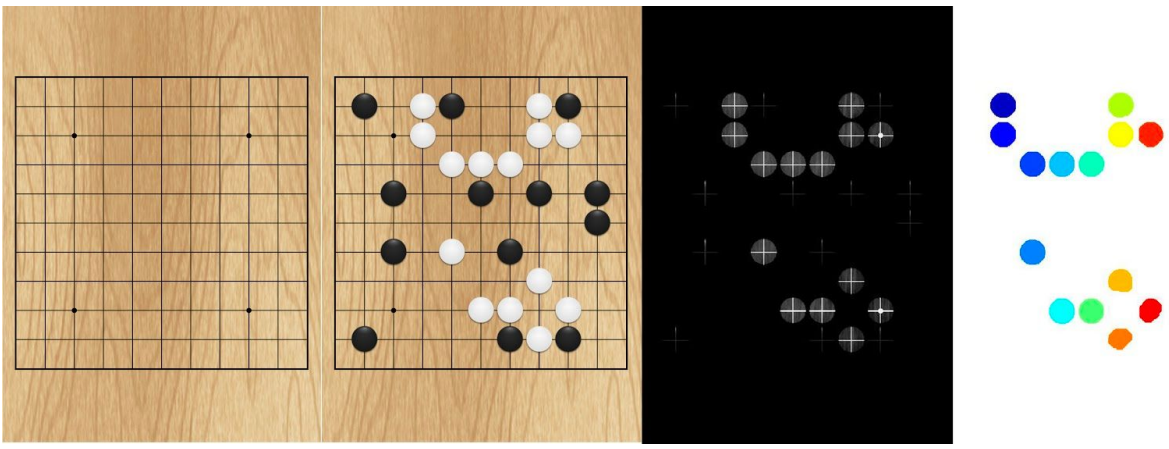

Figure 3. First test images to find the white pieces using a differencing algorithm.

This process was fairly successful in finding pieces for an app. Depending on which image was subtracted from which, would give the different colored pieces.

Some problems with this method was that a plain image of the board was needed, and it would not always be successful in finding pieces of a real photograph. Thus the algorithm was modified to take into account these issues. Furthermore, these images are screenshots of an already digital game, making segmentation cleaner by default.

Second process

With real images of an othello board, the process was much more nuanced. We took our own images, shown in Figure 4. Then, we processed them as described in the algorithm above. Images showing this process can be seen in Figure 5. The abstract representation of the game board is captured in Figure 6, from the Matlab console.

Figure 4. Initial setup for taking real photos of the Othello board.

Figure 5. Raw photo taken through the steps of green_seg code.

```
board =

-------w
--wbb-bb
--bw-wb-
bwwbbbw-
bbwbbw--
b-wbw-bb
bwwbwwb-
-----wbb

White pieces  = 16
Black pieces  = 23
Black is winning!
```

Figure 6. Final abstract representation of the board.

Challenges

Developing our algorithm included meeting the needs of various challenges and even reworking a significant portion of our methodology at one point. Our decision to restrict ourselves to othello saved us from dealing with some particularly difficult challenges. While identifying the type of game would be relatively straight forward (each board looks quite distinct), how the state of each board would be determined would be difficult for the same reason. The boards look very different from each other, so each board would need an entirely different methodology for determining its state. For example, the greenness filter used in othello would not apply to the other two games at all. However, our decision to not work with different games changed this into a non-challenge.

The next major hang up that we hit came around shiny objects. Originally we found a wonderful method call findcircles which seemed to quickly resolve many of our problems. It worked simply and was very effective at finding circles in several of our initial tests. However, over time we began to realize that it was rather flawed for our uses. As it turns out, when the board came close to being full, it's success rate dropped off. We suspect this was due to the camera auto correcting for the lower contrast with so many black pieces. This flaw caused us to abandon find circles all together. Rather than attempting to adjust the inputs to what, to us, was a black box we thought it easier to construct our own method. This led to the algorithm above which uses connected components.

The grid seemed to be a challenge at first. We spent a good deal of time considering different methods for how to tackle the problem. Initially, we felt that working with the contrast of the lines on the board would be our best option. Later we thought about the start state of othello. Othello starts with four pieces placed in the center: two black, two white. The pattern is constant, so extrapolating from those seemed like an option. Eventually we settled on our current solution. By simply using the bounding box of the board it became very easy to create a grid. While the grid is not perfect is a reasonable approximation and any piece placed in vaguely the correct position will have its centroid fall within the correct grid position.

The orientation of the board is still a problem in regards to the grid. The program uses regionprops to find the board. The bounding box that is returned has each of its edges parallel to the edges of the image. As a consequence if the board is rotated within the image, say forty-five degrees, then you get a “diamond in a square” kind of effect. This creates a significant issue when creating the grid, it appears wildly different than the reality.

Determining the color of the pieces proved to be something of a challenge as well. This came about for the same reason that findcircles failed, the black pieces are shiny. As a result they have a significant white component to them. After trying numerous methods it was found that regionprops had the ability to return the intensity of a connected component. This could reliably determine between black and white images based on the mean intensity.

Integration with Othello-playing AI

We did manage to integrate our work (sort of) with a simple AI that plays a digital game of othello.. The code for this project is available on the MathWorks website [10]. The implementation is very finicky and it does not really work for any arbitrary game state (because the AI expects a logical, sequential game of othello), but we were able to integrate it as a proof of concept. That code can be seen in the file `Othello.m`.

Future for this project

This algorithm still has a lot of work that needs to be done before it could be something launched on an app store for people to download and use. Some of the tasks that need to get done are making it easy to use on a phone and able to process the images very fast. The various game boards out there will present a challenge along with all of the different angles that a user could take.

A key flaw of the program is its inability to deal with a rotated board. Due to how detection of the board is done, the edges of the board must be parallel to the edges of the image. If it deviates too far, the program will fail to get the state. Being able to detect and rotate the board into the correct orientation is an important future problem to aid usability.

Using the program with the AI is also rather clumsy. Without a way to automatically take images the program must query the user for the file of each subsequent image. Finding a way to allow matlab to access a camera connected over USB is an important future step.

Bloopers and Interesting Finds

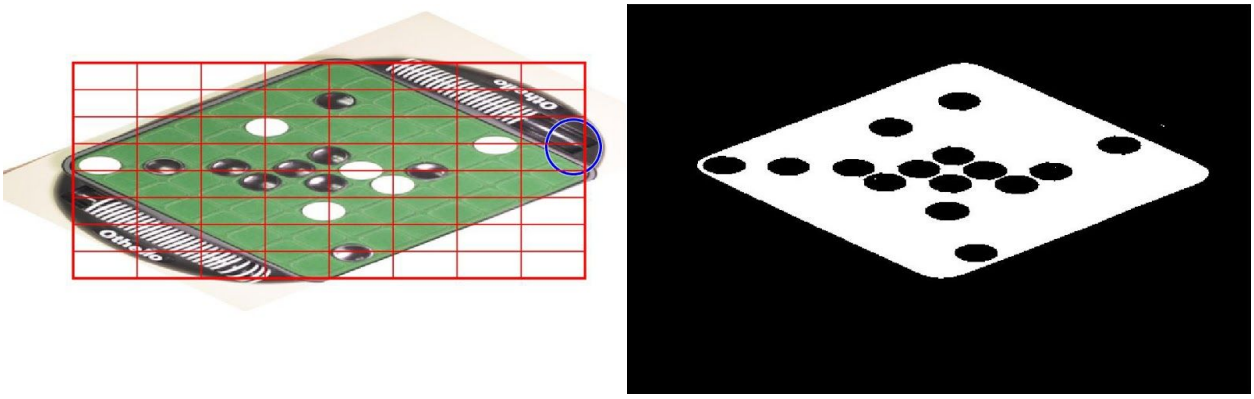

Figure 7. Picture taken off axis began to mess up the algorithm. This was early on in the second process.

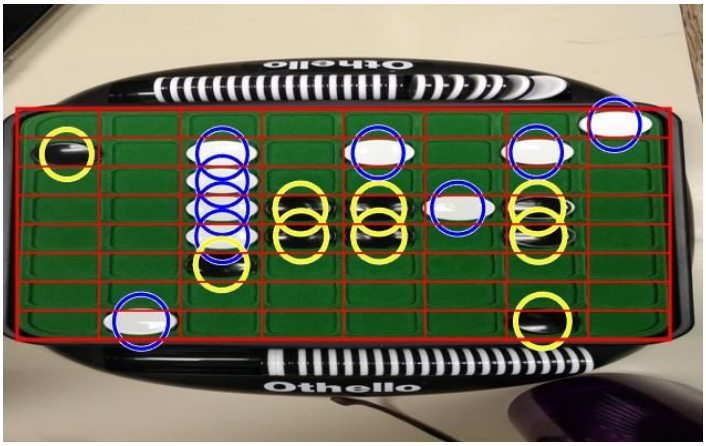

Figure 8. Distorted picture taken from cell phone. Actually works OK.

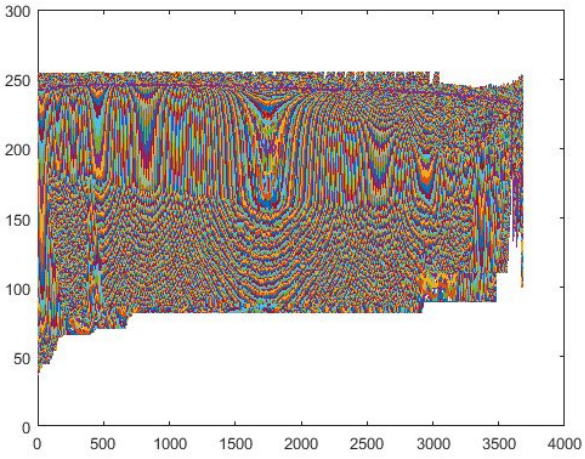

Figure 9. We do not even know what happened for this image, but it is really unique.

Related Work

- Rubik's Cube solver [5]
  - This solver is a good example of detecting a grid of squares, along with analyzing their color.
- Checkerboard Recognition [2]
  - This program talks about a methodology for doing checkerboard analysis. It comments on the challenges of dealing with pieces of a similar color to the underlying square.

- A Computer Vision System for Chess Game Tracking [3]
    - An academic source for an approach to this problem; it provides a good overview of corner detection and position adjustment for a checkerboard
- Image2SGF (SGF - Smart Game Format) [6]
    - Implementation of the main problem statement: interpreting the state of a game of Go and converting it to a Smart Game Format
    - Coded in Perl with terrible documentation, so it won't make this project that much easier
    - Serves as a good reference to one approach to the problem, as well as a link to a more standardized method of representing game states (SGF)

References

[1]-https:.cs//webdocs.cs.ualberta.ca/~chinook/project/

[2]-https://www.bgu.ac.il/~ben-shahar/Teaching/Computational-Vision/StudentProjects/ICBV061/ICBV-2006-1-YuvalFledel-VitalyKhait/checkers_report.pdf

[3]-http://vision.fe.uni-lj.si/cvww2016/proceedings/papers/21.pdf

[4]-https://www.mathworks.com/help/vision/ref/detectcheckerboardpoints.htm

[5]-http://www.vision-systems.com/articles/print/volume-11/issue-12/technology-trends/image-processing/robots-and-vision-solve-rubikrsquos-cube.html

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

[7]-https://deepmind.com/research/alphago/

[8]-http://blogs.mathworks.com/steve/2014/08/12/it-aint-easy-seeing-green-unless-you-have-matlab/

[9]-http://www.cie.co.at/index.php/Publications/index.php?i_ca_id=461

[10]-https://www.mathworks.com/matlabcentral/fileexchange/34989-othello?focused=5224362&tab=example