Sudoku Solver

CS455-Final Report

Nilay Altun

Ben Berger

1. Abstract

The program we developed enables the user to take a picture of a sudoku puzzle and have it solved for them. This method is superior from using an online solve in which the user has to manually enter in the numbers, one at a time. Our main goal with this project was to create digit recognition software.

2. Introduction

Sudoku is the name of a style of puzzle that originated in Japan. It requires a lot of logic skills to solve, and they are very popular. Sudoku puzzles can be complicated to solve, so many online programs have emerged that will solve the puzzle for you. To do so, one must enter in the number given to you by the puzzle one by one. This can be very tedious and monotonous. As visual processing students, we took it upon ourselves to come up with a better solution.

The initial idea was to use a camera lens to take a picture of a puzzle and store it in .jpg form. Afterward, we would operate on the image to find the location of the puzzle in the picture. Then, we would go through each box and check if their was a number present. If so, we would use number recognition software. We would store the numbers in an array, and use a simple sudoku solving algorithm to get the results to the user.

Our main goal with this project was to create digit recognition software. As such, in the very beginning we skipped steps in order to be able to focus on that part of the project. Our hope was that we would be able to come back and work on some of the smaller aspects of the program afterward. Sadly, the digit process implementation took longer than we expected, and were not able to complete the smaller portions of our initial concept.

It should be noted that we are not the first to come up with this problem. When researching the problem, we found that it has been tackled before and can be found in some IOS and android applications. As computer science students, we hoped to develop our own implementation.

3. Preliminary steps

As mentioned above, our main goal with this project was to implement digit recognition. For this reason, we decided to jump straight to the numbers. To do so, we needed a template to work with. For that reason, we searched a website that would provide multiple sudoku puzzles for us to use, along with a printing option. We came across the website, http://www.websudoku.com/, which can be seen below in image 1. It had everything we needed so this became our default.

Pressing the print button gave us an entire page as a .pdf. We had chosen to use opencv with c++ in our implementation, so we needed to convert the puzzle from a .pdf to a .jpg. For that we used the website, http://pdf2jpg.net. Once we had the image as a .jpg we could import it into our program.

From there, all we had to do was determine the region of interest of the actual sudoku puzzle found in the image. Idealy, we would have used the hough transform and edge detection to automatically find the puzzle on the page. Because we were pressed for time, and wanted to complete our main objective, we hard coded the location. We created a Rect ROI (region of interest), applied it to our image to create a modified image as can be seen by image 2.

Lastly we created an algorithm that would loop through each box and determine if a number was present. For this, we created a 9x9 nested for loop, one element for each box. During this operation, each box was thresholded. If a black pixel was found, we knew there was a number in that box and sent it to our getNum function to find out what it was.

4. Methods and Algorithms

At this point, we have a box that contains a number. Because of the thresholding algorithm mentioned above, we know that there is a number present. Now we are ready to begin a series of tests to determine what that number is.

Comparative Overlay

The first method we tried was a comparative overlay. This entailed going to the original template, taking a box that we knew had a number in it, and saving the thresholded box as a template file on the local machine (see image 3). Afterward, we would essentially place that template, on the current box and see if there was a match by checking if the black pixels lined up. If they did, it meant that the number in the box was the same as the number in the template.

There were two problems with this we found. The first one we solved. For whatever reason, we saved the template as a .jpg file. That was stored differently than opencv wanted, and we could never create a match. Changing how we saved the template from .jpg to .bmp solved that problem.

The other problem was that it was impossible to get the number to be in the same location inside the box every time. Because of how the puzzle was set up, when we went to grab the numbers, some might be a little to the right, some a little to the left, and some in different corners of the box. For that reason, simply placing the template over the image would not work.

Counting black pixels

Our next attempt at solving this problem was to count all the black pixels, and match them to stored values. Hopefully we could create some sort of matching scheme to make this process easy. If you recall from above, we are now in a function that has a box with some amount of black pixels in them that arrange to look like a number.

First we went to a box that had a number we knew. We looped through and counted each black pixel, wrote it down, and moved to another box that had a different number. We did this until we had a black pixel count for all numbers, 1 - 9. It was immediately apparent that this method would not work. The first problem was that of the 6 and 9. Those two digits are just the inverse of one another, that counting the black pixels would always result in the same for both.

The other problem only revealed itself when we started testing. For some reason that we were never able to explain, some times the same number would have different black pixel counts. I.e. we would check a four and found it had 142 black pixels the first time, and find a different four that had 140 black pixels.

Morphology

Still on the process of counting the black pixels, we attempted to solve the problem that sometimes the same number would have different numbers of black pixels. To do this, we used morphology.

We tried several methods of erosion and dilation, and found that for erosion with a 5x5 cross kernel, followed by a dilation of 3x3 block, we could get pretty distinct black pixel counts for each number. I.e. each time we found a 4, it would have the same number of black pixels.

This however did not solve our 6/9 problem. They were still outputting the same black pixel count. From here we had two options. The first would be to have our program guess which number it was, test and see if it worked in the puzzle, and try again if it didn't. As we were trying to solve this with visual processing, we elected to not follow that method.

The second option we had was to cut the box that we were working on, and use the halves to figure out which number we had found. The idea was that we would cut the image in half horizontally, and if the top half of the image had a greater black pixel count than the lower, we would know it was a 9, and visa versa. The problem we encountered with this method was that we could not always know where the number was going to appear in the box. I.e. the number 6 could be in the top right corner, and cutting the image in half would result in the top half having a larger black pixel count.

As a note, looking back, it might have been possible to scan the image for the location of the top of the number by finding the first black pixel. From there we could have cut the image a certain amount of lines down and measured from there. This is not what we ended up doing, but it may have worked.

Template, KNN, Tesseract

After much research into the topic, we tried our hand at implementing different functions that were already being used for similar applications. The first one we tried was the template function. This is often used to find a small image in a much larger image. A common example of this is facial recognition. It will take a small image of a face and try and match up with a face in a larger image based on small details. It will return an array of pixels and the ones with the whitest location are usually where the match was found. Unfortunately we tried mutating this use to our program, and found that it wasn't giving the results we wanted. We tried using our templated from the comparative overlay and tried finding matches in the boxes. It didn't work out.

We also tried implementing two learning algorithms, KNN and tesseract. These work by first learning what they are looking for, and then scrubbing the image looking for matches. KNN we could never get working, and when implementing tesseract, we kept getting missing library errors, we we gave up on that. We did make a note to go back to those methods in case our next attempt did not work.

Sliding Template

With our last attempt, we tried implementing the template function from above in our own way. The first thing we did was create templates of each number 1 - 9 and stored them as .bmp's. Then, in our function, we would slide the template over the box (recall that this is a box containing some black pixels which form a number). At each location we would check and see if the pixels lined up. If they did, we would update the confidence interval by 1.

When it finishes with one template, it would check and see if it has the highest confidence interval of any of the templates scanned so far. If it was, then it would update the assumed number based on which template was just scanned. Afterward, it would jump to next template and repeat the process.

At the end, whichever template had the highest confidence interval would be returned as the number that was in that box. Using this method, we were able to achieve 100% accuracy.

This, however, was very slow. So to improve the speed we implemented a few changes. The first was a cap on the confidence interval. If the confidence interval ever reached 90%, the program would return that it had found the right number. We also decreased the area of the image that had to be searched. We found that the edges of the box never contained the number so we could skip those in the search.

Another method to decrease run time was in how we chose which template to run. Instead of running in order, we randomized the order for the search because finding a 2 was just as likely as finding an 8. And if we ran sequentially, if the number was 8, we would check 1-7 beforehand. The second part of this was to remove a number from the search if it had been found in that row. Because sudoku does not allow the same number in a row, we could safely remove that number from ones we needed to check.

5. Results

Using the sliding template method, we were able to achieve 100% accuracy when finding the numbers. Once we did, we threw it into a backtracking sudoku solving algorithm, and printed the numbers into a blank array. Final results are shown below in image 4.

6. Conclusion and Discussion

We were able to accomplish our initial goal of digit recognition. After various methods, and lots of optimizations, the program can solve the sudoku puzzles we give it in about 5 seconds. We can make this even faster by decreasing some of the accuracy, but it doesn't seem worthwhile to do so.

There are many limitations to our code. For example, it works well because we had the template to work with. If we wanted to include more websites, we would have to create more templates, and that would take much longer to run. For that reason, our solution is not great for practical real world applications.

There are a few things we would have liked to implement, and could implement going further. The first is auto finding the puzzle in an image. This would push greatly into more practical applications if the user can just take a picture of the puzzle and the program would find it in the image.

The other thing we would like to implement in the future would be different algorithms for digit recognition. Using methods like KNN and tesseract could push the usage of this program into real world solutions. They could help with situations such as human handwriting, something our program would never be able to handle.

7. References

(1) Realtime Webcam Sudoku Solver, Bank B., 2011.

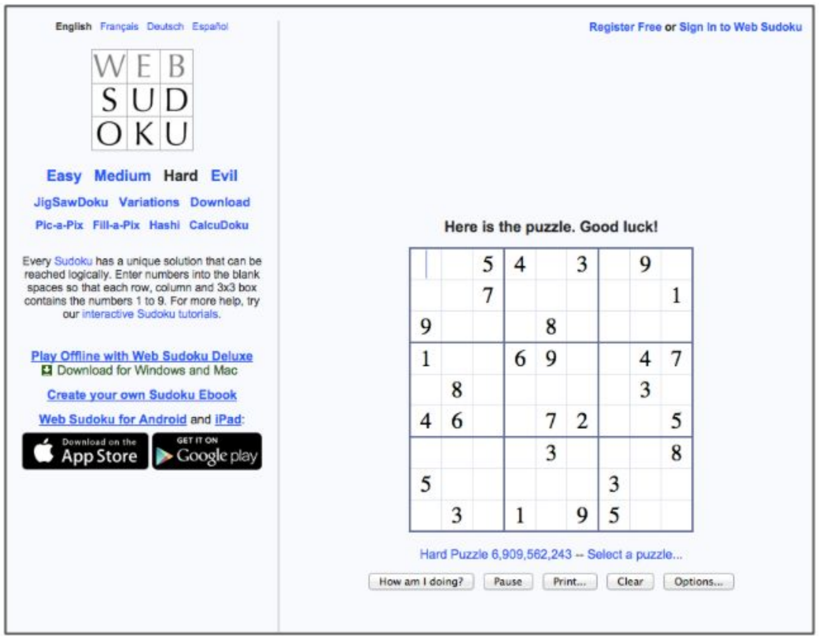
http://www.codeproject.com/Articles/238114/Realtime-Webcam-Sudoku-Solver

(2) Sudoku Solver Android App, Wang, Y., 2014, *Stanford University*

http://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Wang.pdf

(3) C++ BackTracking, M. Bhojasia, 2013

http://www.sanfoundry.com/cpp-program-solve-sudoku-problem-backtracking/



English Français Deutsch Español
Register Free or Sign In to Web Sudoku
WEB SUDOKU
Easy Medium Hard Evil
JigSawDoku Variations Download
Pic-a-Pix Fill-a-Pix Hashi CalcuDoku
Play Offline with Web Sudoku Deluxe
Download for Windows and Mac
Create your own Sudoku Ebook
Web Sudoku for Android and iPad:
Download on the App Store
GET IT ON Google play
How am I doing?
Pause
Print...
Clear
Options...

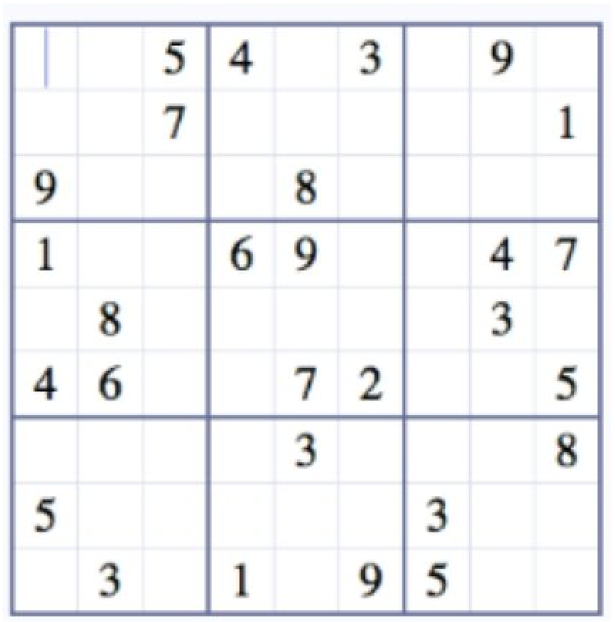
Every Sudoku has a unique solution that can be reached logically. Enter numbers into the blank spaces so that each row, column and 3x3 box contains the numbers 1 to 9. For more help, try our interactive Sudoku tutorials.

Here is the puzzle. Good luck!

		5	4		3		9	
		7						1
9				8				
1			6	9			4	7
	8						3	
4	6			7	2			5
				3				8
5						3		
	3		1		9	5		

Hard Puzzle 6,909,562,243 -- Select a puzzle...

Image 1



		5	4		3		9	
		7						1
9				8				
1			6	9			4	7
	8						3	
4	6			7	2			5
				3				8
5						3		
	3		1		9	5		

Image 2

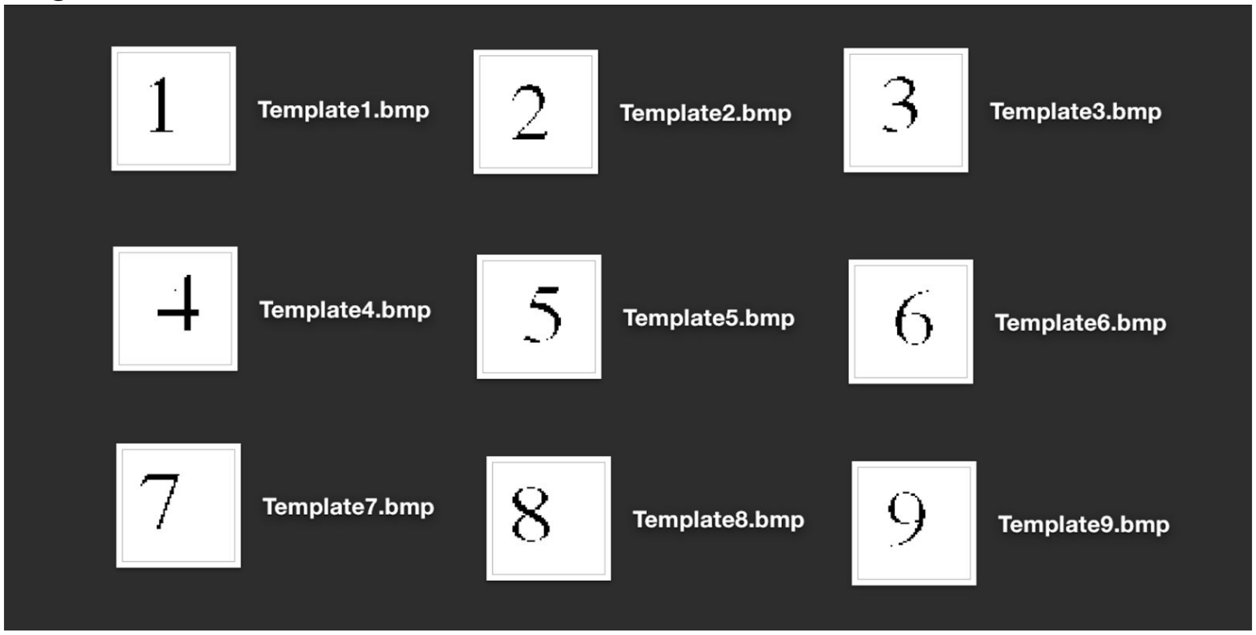
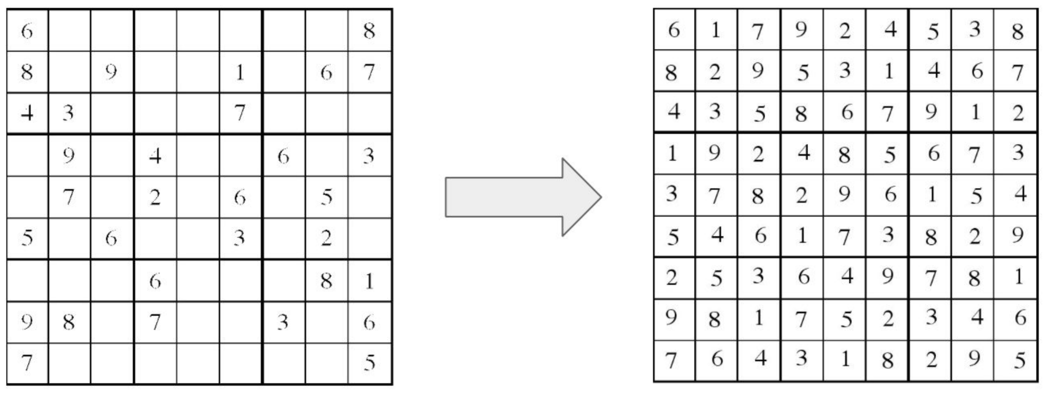


Image 3



6								8
8		9			1		6	7
4	3				7			
	9		4			6		3
	7		2		6		5	
5		6			3		2	
			6				8	1
9	8		7			3		6
7								5

6	1	7	9	2	4	5	3	8
8	2	9	5	3	1	4	6	7
4	3	5	8	6	7	9	1	2
1	9	2	4	8	5	6	7	3
3	7	8	2	9	6	1	5	4
5	4	6	1	7	3	8	2	9
2	5	3	6	4	9	7	8	1
9	8	1	7	5	2	3	4	6
7	6	4	3	1	8	2	9	5

Image 4