Group Assignment: Sudoku Solver

Image Processing MECH-B-5ANT-ANT-ILV

WS 2024 Image Processing MECH-B-5-MRV-IMP-ILV

Bachelor's degree - Mechatronics, Design and Innovation

5th Semester

Lecturer: Daniel McGuiness, PhD

Group: BA-MECH-22

Author: Thöni Andreas, Samuel Peer, Hannes Unterhuber

Student Number: 52216067, 62200066, 61901292

December 4, 2024

Contents

1 Introduction

The goal of this assignment was to create a program to unwarp a picture of a Sudoku field. This imposed a few challenges the team had to overcome:

- Correctly detect a sudoku field from images
- De-strech and un-warp the puzzle to create a rectangular image

In addition, the team decided to go beyond this goal by using optical character recognition (OCR) to recognize the numbers inside the of the grid and implementing an algorithm to solve the Sudoku. All of these features have been made usable inside of a desktop application. Puzzles can be loaded using the filebrowser or by using a camera. The goals were defined as follows:

- Recognizing different sections of the field
- Recognizing numbers
- Allocating the correct numbers to their respective position in the matrix
- Implementation of an algorithm to actually solve the puzzle
- Overlay of the solution on the unwrapped picture

The tasks were divided among the team like this:

- The recognition and de-stretching of the Sudoku grid and its sections was handled by Andreas Thöni.
- The number recognition using OCR was implemented by Samuel Peer.
- The development of the applet, the GUI, and the integration of the overlay was carried out by Hannes Unterhuber.

Source code repository: https://github.com/ajayhanssen/SudokuSolver

2 Implementation

This chapter discusses the way the code works from a qualitative perspective, each part being made by the corresponding group member.

2.1 RECOGNIZING GRID AND DE-WARPING

This part of the programming was implemented by Andreas Thöni. The first approach to recognizing the sudoku grid was to use the Hough-Line-Transform algorithm, as it was being recommended by the lecturer as a good starting point. Attempts trying to get the program to work using this method can be found in "houghline.py". These attempts, however, led into a dead-end and were abandoned, because precise adjustments of the parameters would be necessary and not feasible. Instead, the OpenCV-function "findContours()" was used for the final version. It can be found in the file "geometrical.py". The main focus lies on the "un_warp_sudoku()"-function. It takes an image as an argument, a warped picture of a sudoku puzzle in our case. The way it works is this:

1. The input image is converted to grayscale as we do not need the colour channels and it simplifies the process tremendously.
2. Gaussian blur is applied to make the edge detection (using canny edge detection) that is following easier.
3. Canny edge detection is used to detect hard edges on the image.
4. "findContours()" is used to convert the edges from before to contour-information (Lists of points).
5. The detected edges are being sorted by the size of their area, beginning with the largest one.
6. By iterating through every contour that was detected and using OpenCV's approxPolyDP() function to approximate the contours as polygons, the first polygon with 4 edges, which would represent the outmost border of the sudoku, could be saved for further processing.
7. The next step was to prepare the information from the functions from earlier for dewarping. For this, two arrays needed to be defined. The first one of which contains the locations of the four edges of the new image (a width and height of 450 was chosen), in our case looking like this:

$$dst = [[0,0],[449,0],[449,449],[0,449]] \tag{1}$$

   and the corresponding location of the 4-polygon-contour that was detected earlier, also sorted clockwise (top left, top right, lower right, lower left).
8. Using OpenCV's "getPerspectiveTransform() with those two arrays as arguments (origin points and target points), it was possible to calculate a perspective transform matrix that could be used to straighten out the image.
9. The transformation matrix was applied to the image using the function "warpPerspective()", which resulted in the image being unwarped. The steps have been visualized in figure 2.1.

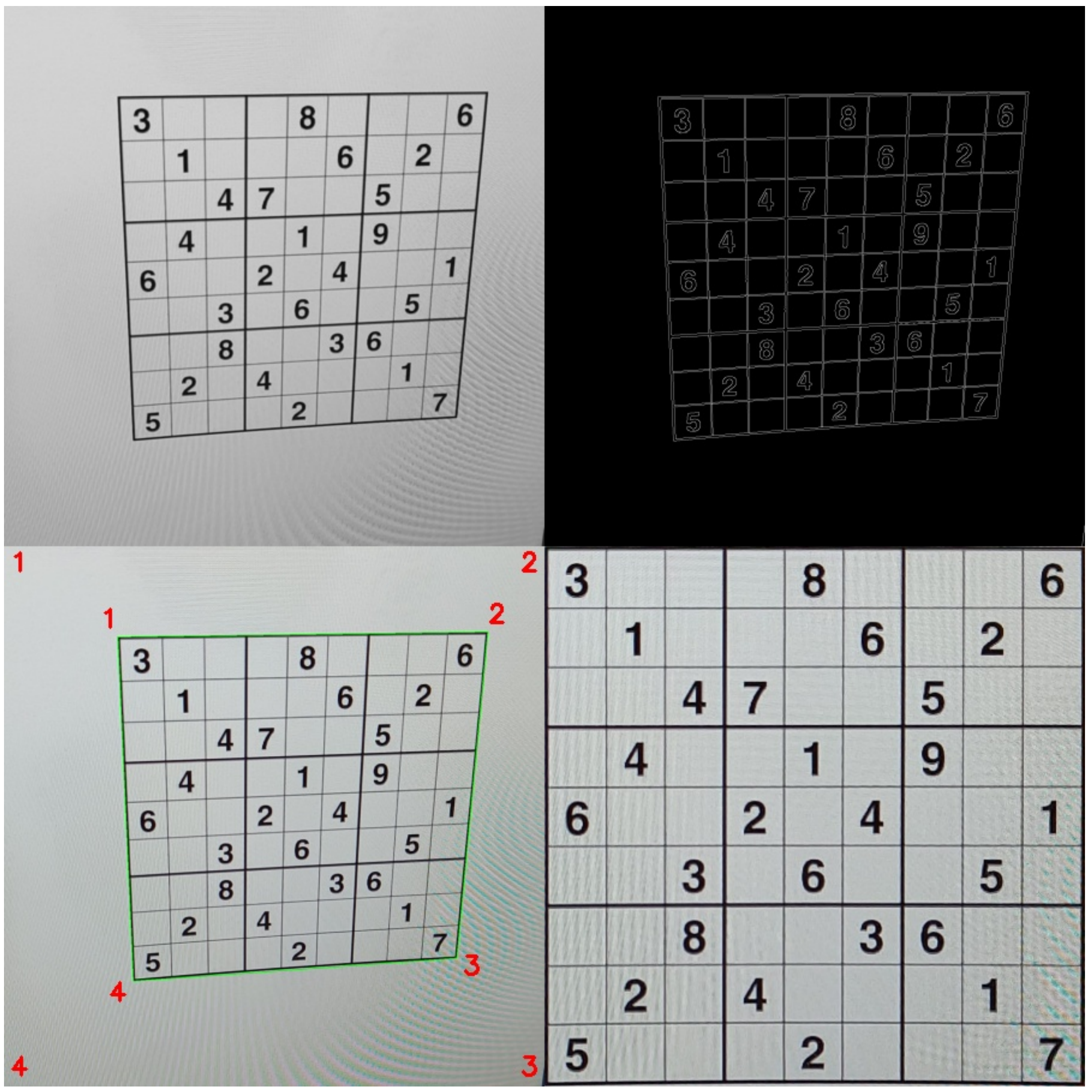

Figure 2.1: Progression of sudoku detection process from top left clockwise: Blurred original image; applied canny-edge-detection; detected outmost contour and transformation points; final, de-strechted image

The function can now be used to automatically detect, de-warp and re-scale any image containing a visible sudoku puzzle on it.

2.2 NUMBER RECOGNITION

This part of the project was completed by Samuel Peer. Our next goal after un-warping the image was to correctly recognize the occuring numbers in the grid and turn them into a 9×9 matrix, which is needed to solve the puzzle later on. To achieve this, some preprocessing steps were necessary, as well as splitting the puzzle into individual cells. Some Gaussian blur was applied once again and the image was binarized with a certain threshold to make the number recognition itself easier. A comparison of a single cell between before and after the operations can be seen in figure 2.2.

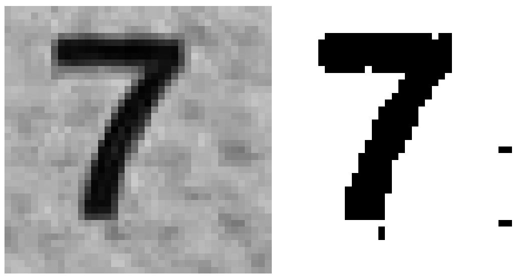

Figure 2.2: Singular grid cell before and after preprocessing

After preparing the image, the digits within the cells are extracted using OCR (tesseract):

1. Pytesseract's OCR processes each cell with a configuration tailored for single-character recognition:

   ```
   number = pytesseract.image_to_string(cell, config='--psm 10 digits')
   if number in valid:
       board[i][j] = int(number)
   ```

2. The recognized numbers are organized into a 9x9 matrix for further use by the solving algorithm.

These steps resulted in a function called "construct_board()", which is located in "field_recognizer.py"

2.3 SOLVING ALGORITHM

The code works with the brute force method.

Listing 1: Sudoku Solver Code

```
import numpy as np

def is_valid(board, row, col, num):
    if num in board[row,:]:
        return False

    if num in board[:,col]:
        return False

    box_row = row // 3 * 3
    box_col = col // 3 * 3

    if num in board[box_row:box_row+3, box_col:box_col+3]:
        return False

    return True

def find_empty(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return (i, j)
    return None

def solve(board):
    empty = find_empty(board)
    if not empty:
        return True
    else:
        row, col = empty

    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num

            if solve(board):
                return True

            board[row][col] = 0

    return False
```

2.3.1 Key Functions

- `is_valid(board, row, col, num)`: This function checks whether placing a number (`num`) in a specified cell (`row, col`) of the board is valid. It ensures that the number does not already appear in the row, column, or 3×3 subgrid of the cell.
- `find_empty(board)`: This function searches for an empty cell in the board (represented by the value 0) and returns its coordinates as a tuple (`row, col`). If no empty cells exist, it returns `None`.
- `solve(board)`: This is the main recursive function implementing the backtracking algorithm. It attempts to solve the puzzle by placing a valid number in an empty cell and recursively solving the rest of the board. If placing a number leads to a conflict, it backtracks by resetting the cell to 0 and trying the next number.

2.3.2 How the Program Works

1. The program begins with a partially filled Sudoku board represented as a 9×9 NumPy array.
2. The `solve` function is called to attempt solving the puzzle.
3. Inside `solve`, the program identifies an empty cell using `find_empty`.
4. It iterates over numbers 1 through 9, checking if each number is valid using `is_valid`.
5. If a valid number is found, it is placed in the cell, and the program recursively attempts to solve the rest of the board.
6. If no valid number can be placed, the program backtracks by resetting the cell to 0 and continues trying other numbers.
7. The process repeats until the board is completely filled or determined unsolvable.

2.4 GRAPHICAL USER INTERFACE

2.4.1 Menu description

The GUI was implemented by Hannes Unterhuber using the "tkinter" library and the "ttkbootstrap" extension for improved aesthetics. The interface includes:

- A home menu for navigation (Figure 2.3).
- A camera interface for live puzzle capture (Figure 2.4).
- An option to load and solve saved puzzles (Figure 2.5).

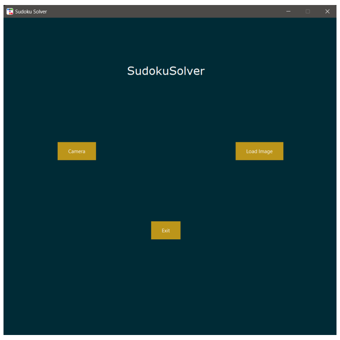

Figure 2.3:
Homepage

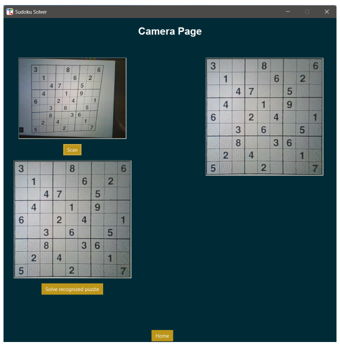

Figure 2.4:
Camerapage

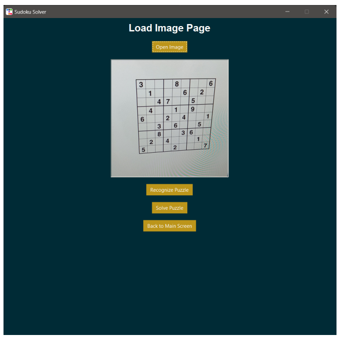

Figure 2.5:
Loadingpage

2.4.2 Class Descriptions

- **SudokuSolverApp class**: Manages the overall user interface and transitions between pages.
- **MainPage class**: Provides navigation to different features and displays credits.
- **CamPage class**: Captures puzzles from a live video feed and overlays solutions.
- **LoadPage class**: Allows users to upload images for recognition and solving.

Both the CamPage and the LoadPage classes utilize the same algorithm to unwarp the picture and detect the puzzle 2.6. By pressing the solve button and allowing pytesseract a brief calculating time, the puzzle will then be solved, and the solution will be ouput with a red overlaying number for each corresponding field2.7. Should there be no solution or an error in the recognition process the programms command line will output the errormessage.

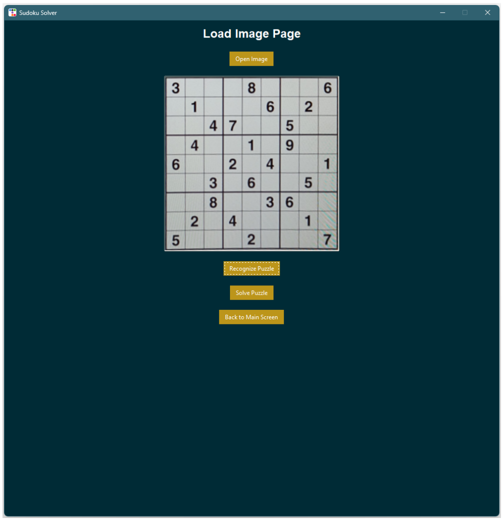

Figure 2.6:
Detected and unwarped picture

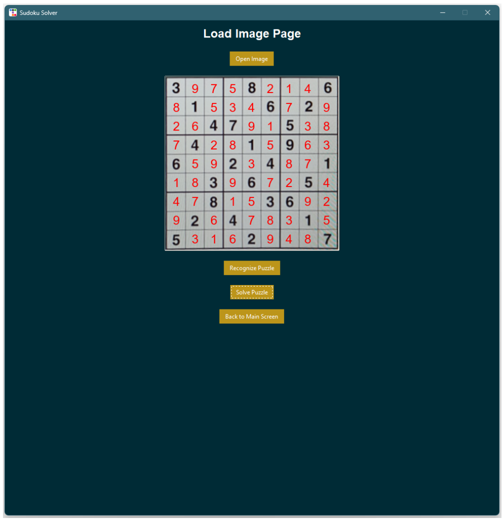

Figure 2.7:
Puzzle with solution overlay