SUDOKU SOLVER

Andronescu Raluca
Image Processing
Grpou 30431
2025

DESCRIPTION

This project is an Automatic Sudoku Solver that uses image processing to recognize and solve Sudoku puzzles from scanned or photographed images.
It detects the grid, extracts individual cells, and uses custom thresholding and template matching to recognize the digits in each cell. Then, solves it using a backtracking algorithm, and displays the complete solution in a clean, readable format.

Input

A scanned or photographed image (JPG, PNG, or BMP) containing a single, clear Sudoku puzzle grid.

Output

A digital display of the recognized and solved Sudoku puzzle.

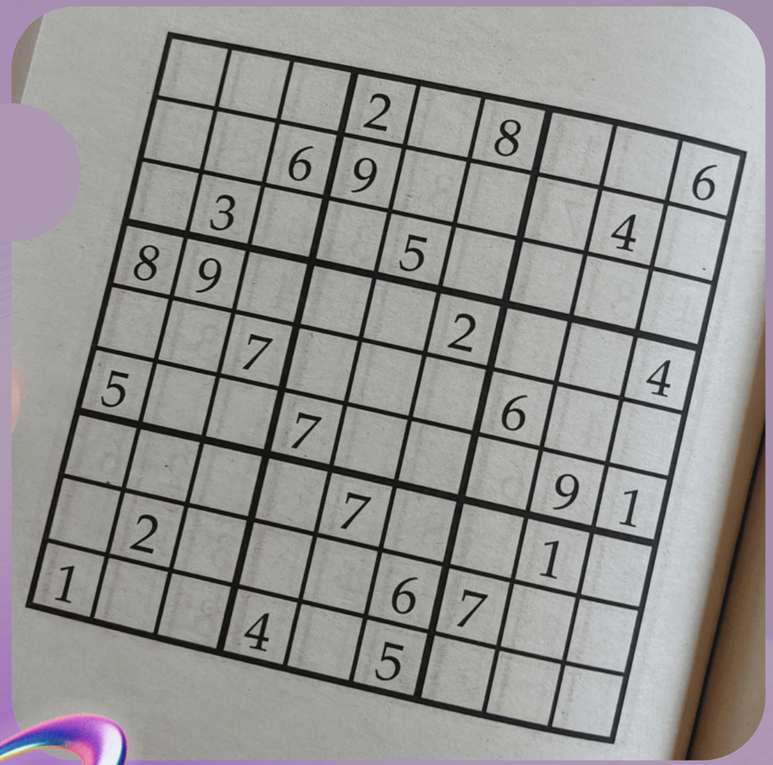
Process

Image preprocessing, grid detection, cell segmentation, digit recognition, and Sudoku solving.

Goal

Fully automate the reading and solving of printed Sudoku puzzles, handling typical challenges like noise, varying fonts, and imperfect scans.

CONSTRAINTS ON INPUT



- The image must not be too dark (digits and grid lines should be clearly visible).
- The Sudoku grid should not be upside down or rotated; it should be aligned upright.
- The grid should be complete and contain only one Sudoku puzzle.
- Digits should be clear and printed in a standard font.

RELATED WORK & DOCUMENTATION

Related Projects

- Sudoku solvers often use classic computer vision or machine learning for grid and digit detection. Notable open-source examples include solutions based on OpenCV (template matching, contour detection) and KNN or CNN models for digit OCR.

Digit Recognition

- KNN (K-Nearest Neighbors): Matches digit images to labeled samples and uses majority voting for recognition.
- CNN (Convolutional Neural Network): Deep learning models that learn features from training data for robust digit classification.

Sources & Links

- https://github.com/guillaumemilitello/sudoku_solver/blob/master/OCR.cpp
- https://github.com/neeru1207/AI_Sudoku/tree/master?tab=readme-ov-file
- https://golsteyn.com/writing/sudoku

HIGH LEVEL ARCHITECTURE

- **Input Image**
  - User provides a scanned or photographed Sudoku puzzle.
- **Preprocessing**
  - Convert to grayscale, blur, and threshold for noise reduction and contrast enhancement.
- Grid Detection & Localization
  - Detect the outer Sudoku grid using edge detection, contour finding, and perspective transformation.
- **Cell Segmentation**
  - Divide the detected grid into 81 individual cell images.
- **Digit Recognition**
  - For each cell, remove borders/noise, center/scale the digit, and match to template images.
- **Sudoku Solver**
  - Solve the digitized puzzle using the backtracking algorithm.
- **Result Display**
  - Show the recognized and solved grid digitally (on-screen or as an image).

IMPLEMENTATION

Loading the Sudoku Image

The program allows the user to select and load a scanned or photographed Sudoku puzzle image from the computer using a file dialog. The selected image is loaded into memory with OpenCV and resized if needed for easier processing.
NEXT STEP : Localize Sudoku

```
void loadSudokuImage(Mat& sudokuImage, vector<Mat_<uchar>> digitTemplates)
```

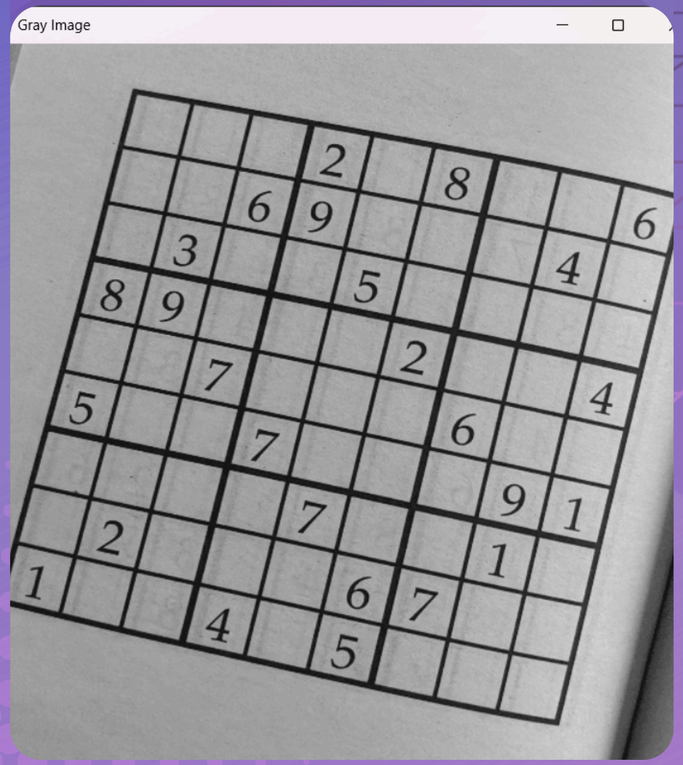
```
Sudoku solver:
1. Load Sudoku Image
0. Exit
Option:
```

```
file_path = open_file_dialog()
input_image = imread(file_path)
if input_image is too large:
    resize(input_image)
```

LOCALIZE SUDOKU

`Mat_<uchar> localizeSudoku(Mat& sudokuImage)`

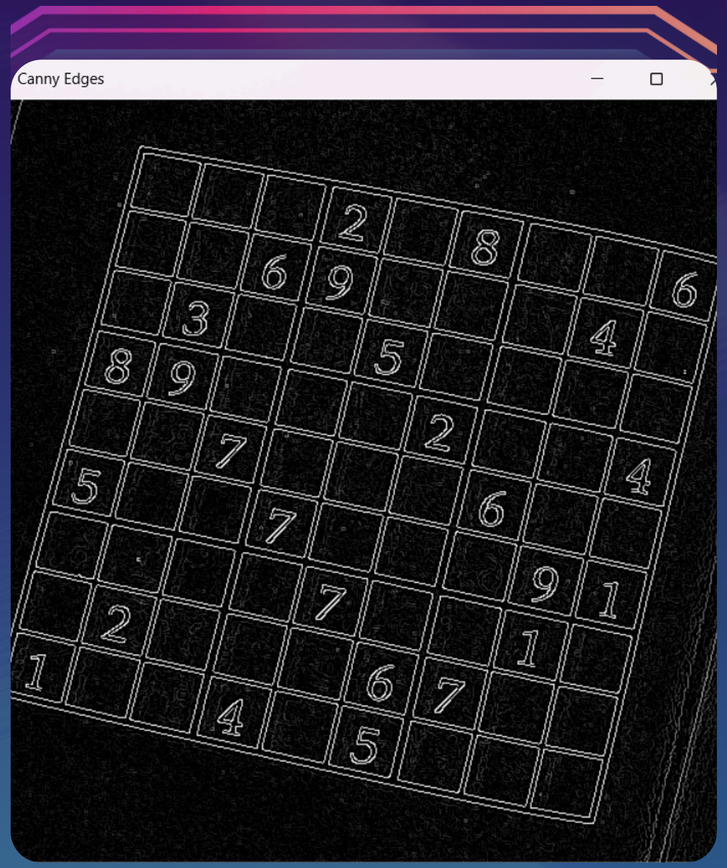
- In order to crop and extract the Sudoku grid from the input image, there are several key steps to follow:




```
gray = to_grayscale(input_image)
edges = canny_edge_detection(gray)
linked_edges = edge_linking(edges)
dilated = dilate(linked_edges)
contours = find_all_contours(dilated)
biggest = select_largest_quadrilateral(contours)
warped_grid = perspective_transform(input_image, biggest)
```

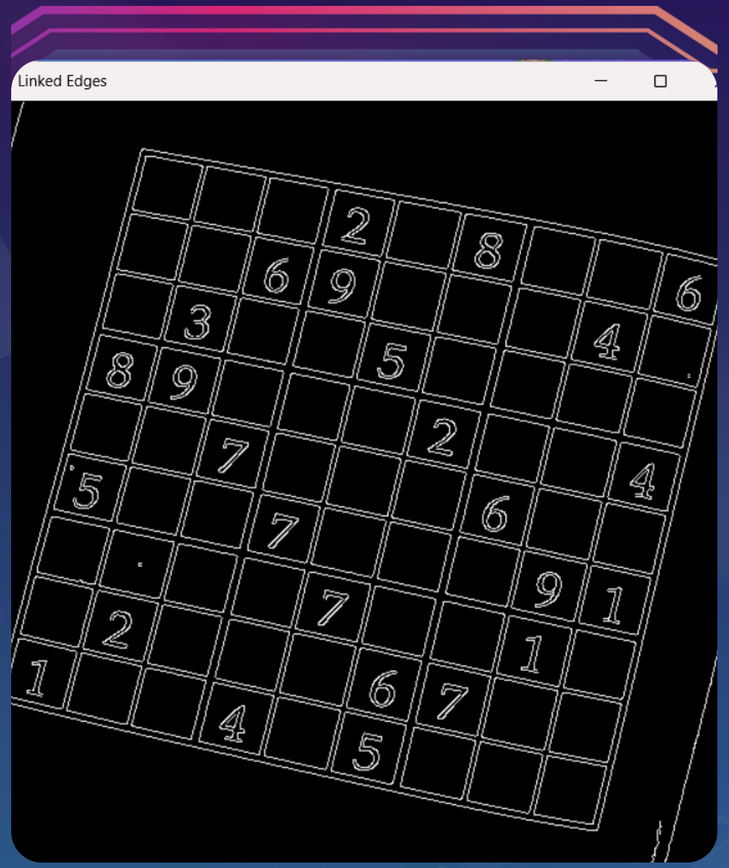
1. CONVERT TO GRAYSCALE

The input color image is converted to grayscale to simplify further processing.



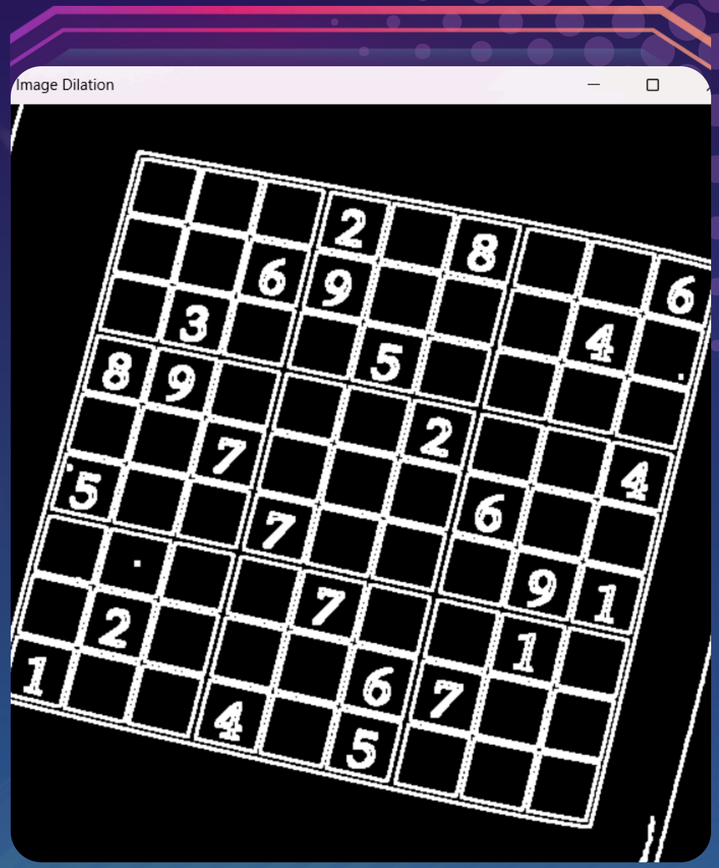
2. CANNY EDGE DETECTION

Detects strong edges, making grid lines stand out.



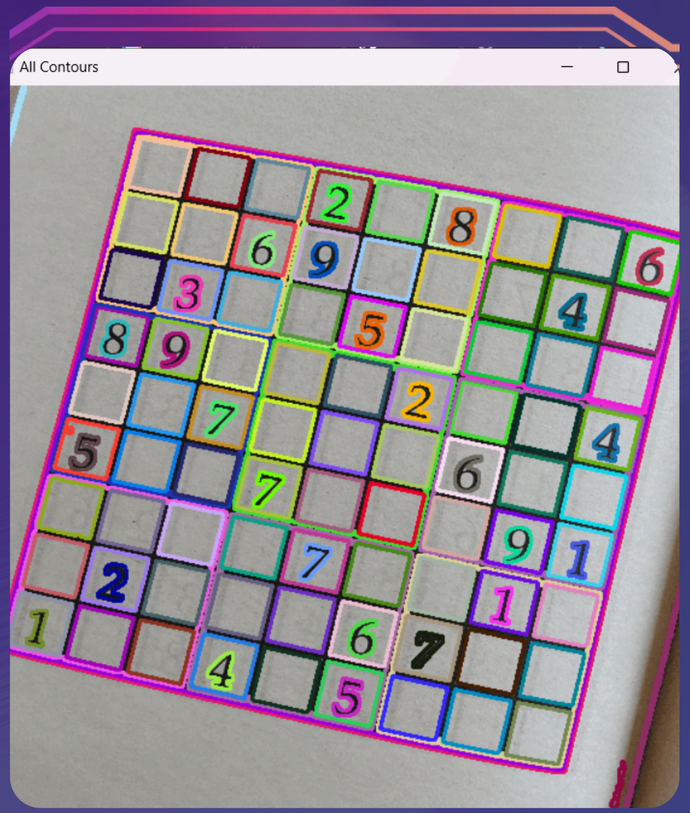
3. EDGE LINKING

Connects edge fragments for cleaner contours.



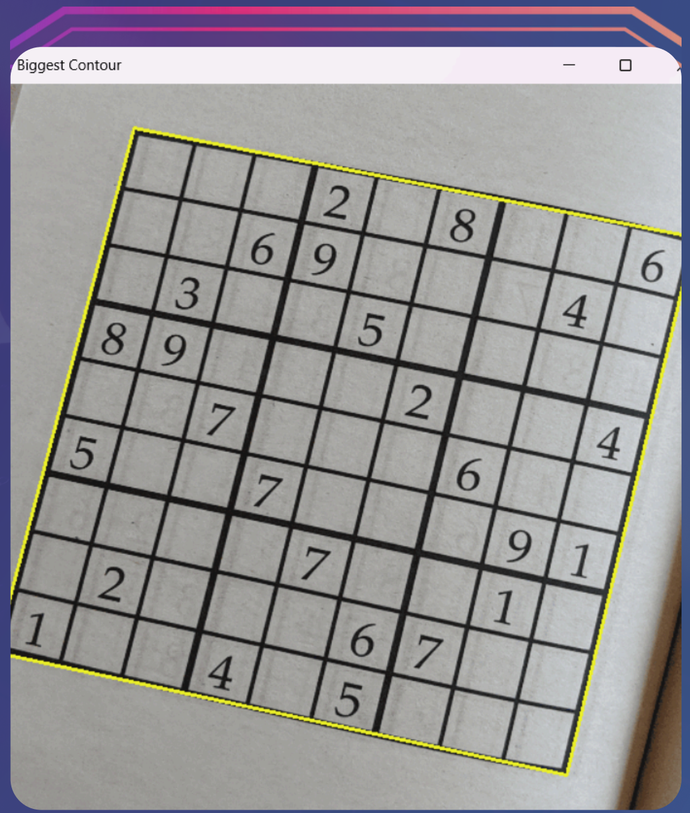
4. DILATION

Expands the edges to strengthen and close grid lines.



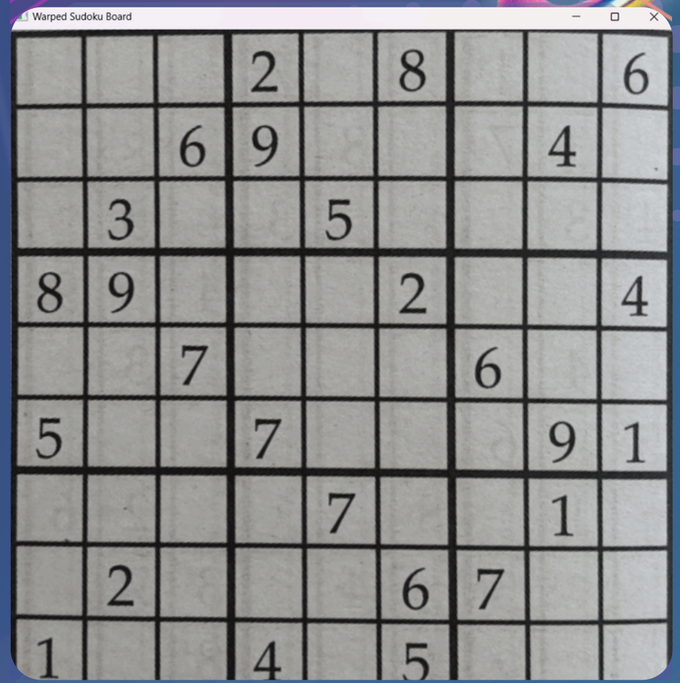
5. FIND ALL CONTOURS

Detects all possible closed shapes in the image.



6. SELECT BIGGEST CONTOUR

Finds the largest 4-point contour, presumed to be the Sudoku grid.



7. WARP SUDOKU

Straightens and crops the grid to a square for further processing.

DIVIDE CELLS

1.CELL SEGMENTATION

After localizing and warping the Sudoku grid, the image is divided into 81 equal-sized cells for digit recognition.

2.CELL THRESHOLDING

Convert each cell image to a clean black-and-white (binary) format, making digits stand out from the background.

3.BORDER REMOVING

Remove black grid lines or artifacts around the edges of each cell, ensuring only the digit remains for recognition.

4.DIGIT CENTERING

Center the digit within the cell image to improve template matching accuracy.

Templates used for comparison contain black digits that fill 80% of the image width. Recentering the image ensuring consistent alignment and scale.

DIGIT RECOGNITION

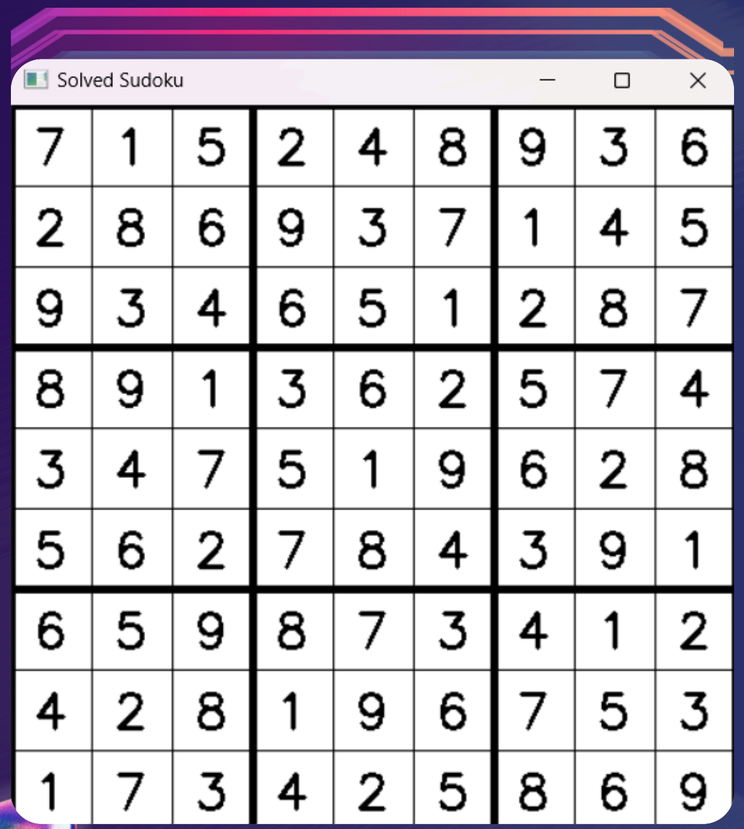
Template Matching

Identify which digit is present in each cell by comparing the processed cell image with a set of labeled reference images (templates) for each digit (1–9).

- For each cell, the algorithm compares its pixels to each digit template of the same size(different fonts).
- It counts how many black pixels overlap match.
- The digit whose template has the highest overlap score (often using the Dice/F1 coefficient for fairness) is selected as the recognized digit for that cell.

```
pixelMatchScore(cell_image, template_image):
   for each pixel (y, x) in cell_image:
       if cell_image[y, x] is black and template_image[y, x] is black:
           match_count ++
       if cell_image[y, x] is black:
           cell_black ++
       if template_image[y, x] is black:
           template_black ++
    score = 2 * match_count / (cell_black + template_black)
```

RESULTS



- The project was tested on multiple scanned and photographed Sudoku puzzles with various print qualities and fonts.
- For each input image, the program successfully localized the grid, recognized the digits, and displayed both the recognized and solved grids.
- Sample cases demonstrate accurate digit extraction and robust solving, even when input images have minor noise or grid imperfections.

USER MANUAL

Instructions

- **Prepare Templates:**
  - Place digit template images (nr_1.png, nr_2.png,...) in a folder named Digits in the project directory.
- **Prepare Input Images:**
  - Place your scanned or photographed Sudoku images in an Images folder.
- **Start the Program:**
  - Run the executable (or start from your IDE).
- **Load a Sudoku Image:**
  - When prompted, select an image file from the Images folder using the file dialog.
- **View Results:**
  - The recognized grid and the solved Sudoku will be displayed in separate windows.
  - The recognized digits are shown in the console and as an image.
- **Exit**:
  - Choose the exit option in the program menu or close the window.

Requirements:

- C++ compiler (Visual Studio or g++)
- OpenCV library (version 3.x or newer)
- Image files: Sudoku scans/photos in JPG, PNG, or BMP format
- Digit templates (PNG images in a /Digits/ folder)

Tips:

- For best results, use clear, well-lit images with the grid upright and digits readable.
- Make sure the template digit images match the font style of your Sudoku puzzles.

CONCLUSION

Results Analysis

The automatic Sudoku solver reliably processes and solves scanned or photographed puzzles using classical image processing and template matching.

The solution is robust for clear, upright grids with standard fonts, and handles most common image imperfections.

Limitations: Recognition accuracy may decrease for very dark, blurry, rotated, or handwritten puzzles. The system is optimized for printed grids with visible digits.

THANK YOU!