CO4204 Computer Vision and Image Processing

**Mini Project - Sudoku Solver**

|  |  |  |
| --- | --- | --- |
| Group Members: |  |  |
| H.A.S.H. Hettiarachchi | 19/ENG/031 | EN |
| K.A.M.D.Kariyawasam | 19/ENG/049 | EN93883 |
| T.T.K.Wijethunge | 19/ENG/117 | EN93927 |

Contents

[1. Introduction 2](#_Toc155041001)

[2. Background and Constraints 2](#_Toc155041002)

[Background 2](#_Toc155041003)

[Neural Network in digit\_model.h5 file 3](#_Toc155041004)

[Dancing Links algorithm 3](#_Toc155041005)

[Constraints 3](#_Toc155041006)

[3. Implementation of the Algorithm 3](#_Toc155041007)

[4. Optimization techniques 4](#_Toc155041008)

[5. Challenges Faced 5](#_Toc155041009)

[6. Limitations 5](#_Toc155041010)

[7. Future Improvements 5](#_Toc155041011)

# Introduction

Sudoku, a popular logic-based number-placement puzzle, has captivated puzzle enthusiasts for decades. Its grid-based structure and simple rules conceal a complexity that challenges and entertains the players.

This particular project represents a convergence of algorithms and computer vision, aiming to provide a robust and versatile solution for Sudoku solving. By harnessing the strengths of Python programming language, neural networks, and the Dancing Links algorithm implemented in C++, our approach strives for efficiency and adaptability.

The integration of computer vision in Sudoku solving marks a transformative leap in puzzle-solving methodology. Using image processing techniques, our algorithm has the ability to "see" and interpret Sudoku grids embedded within images. This intricate process between computer vision and algorithmic optimization starts with the extraction of puzzle grids from images, transforming them into a format conducive to precise algorithmic analysis.

Through this symbiotic relationship, computer vision is expected to enhance the accuracy of digit recognition, ensuring the seamless translation of visual data into algorithm-friendly inputs. This holistic approach not only facilitates the solving process but also fosters a more intuitive and user-friendly experience for individuals interacting with Sudoku puzzles.

# Background and Constraints

## Background

The background of the Sudoku solving algorithm encompasses the combination of computer vision and machine learning methodologies. In this system, a pre-trained neural network, specifically a Convolutional Neural Network (CNN), is employed to recognize and extract digits from the Sudoku grid. This neural network, encapsulated in the 'digit\_model.h5' file, has undergone training to efficiently identify numerical characters within the context of the puzzle. For solving the Sudoku puzzle, the algorithm leverages C++ code that implements the Dancing Links algorithm. This algorithm serves as an efficient technique for solving exact cover problems, making it particularly suitable for Sudoku puzzles.

### Neural Network in digit\_model.h5 file

The model used is a Convolutional Neural Network (CNN) with multiple layers, including convolutional layers, activation layers (using the Rectified Linear Unit or ReLU activation function), max-pooling layers, flattening layers, dense layers, and dropout layers. The model is designed for image classification, as suggested by its input layer's batch shape of [null, 28, 28, 1], which is typical for grayscale images of size 28x28. Additionally, the optimizer Adam optimizer is used and the model is built using TensorFlow version 2.4.0 and mentions classes like "Conv2D," "Dense," and "Activation," which are associated with TensorFlow's Keras API.

### Dancing Links algorithm

The Dancing Links algorithm, devised by Donald E. Knuth, is a specialized technique for implementing efficient backtracking algorithms to address exact cover problems. Utilizing a doubly linked list, the algorithm compactly represents constraints and possibilities. Columns in the list signify constraints, and nodes within columns denote potential choices satisfying those constraints. Initially, the algorithm constructs a matrix representation of the exact cover problem, with columns as constraints and rows as potential satisfying choices. "Dancing Links" symbolizes the selective covering and uncovering of nodes in the matrix during exploration. In Sudoku solving, the algorithm adeptly manages puzzle constraints by efficiently navigating combinations of numbers in rows, columns, and submatrices. It excels in exhaustive searches, pruning the search space during backtracking. The algorithm's use of linked lists allows for a concise implementation.

## Constraints

* Grid Size: The algorithm supports both standard 9x9 Sudoku grids. Additionally, it can handle larger 16x16 Hexadoku grids.
* Input Source: Images can be provided either from files or captured directly from a camera.
* Sudoku Puzzle Representation: The Sudoku grid is represented as a matrix. The grid can be of varying sizes (9x9 or 16x16) depending on the puzzle.
* Input Handling: The main function reads the input Sudoku puzzle from a file. The program checks the size of the puzzle and selects the appropriate solving approach (9x9 or 16x16).
* Output: The initial puzzle grid is printed. The solved puzzle is output as a result of the algorithm's execution.

# Implementation of the Algorithm

**Language**: Python

**Libraries**

* **os**: The os library is utilized to manage file and directory operations, including creating folders, removing files, and checking for file existence, ensuring efficient organization of program output and intermediate files.
* **cv2 (OpenCV)**: OpenCV is employed for various image processing tasks such as contour detection, adaptive thresholding, and perspective transformation, enabling the extraction and manipulation of information from images.
* **numpy**: NumPy is leveraged for efficient array manipulation and mathematical operations, crucial for handling pixel values and conducting numerical operations on image data.
* **operator**: The operator module is specifically used with the sorted function to sort the corners of the Sudoku grid based on their y-coordinates, aiding in the correct order for perspective transformation.
* **subprocess**: The subprocess module facilitates the compilation and execution of a C++ program (dancingLinks) for solving Sudoku, enabling communication with the external program and capturing its output.
* **keras.models.load\_model**: The load\_model function from the Keras library is employed to load a pre-trained neural network model (saved in the HDF5 format) for digit recognition, enabling the identification of digits within the Sudoku grid from images.

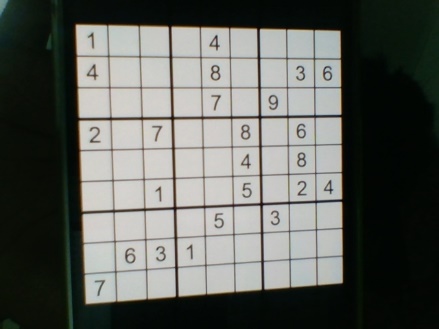
**Steps**

1. Main Execution Loop:

* Continuously present a menu to the user with options to select an image from files, capture an image from the camera, or quit the program.

1. User Interface Functions:

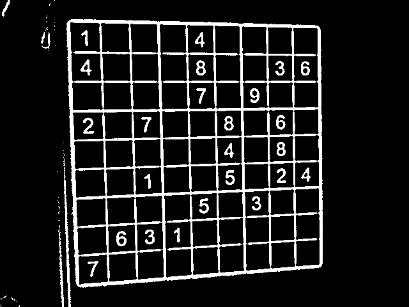
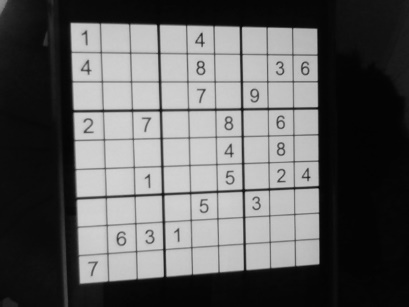
* **select\_image\_from\_files():** Allow the user to input an image file path and process it.
* **capture\_image\_from\_camera():** Enable capturing an image from the webcam and process it.

****

*Figure 01: Captured Image*

1. **extract\_grid(frame):** extract the Sudoku grid from the image

* **binary\_image(frame):** Convert the image to a binary image
  + **cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY):** Convert the image to grayscale
  + **cv2.GaussianBlur(gray, (7, 7), 0):** Apply Gaussian blur to the image to remove noise
  + **cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY\_INV, 9, 2):** Apply adaptive thresholding to get a binary image

****

*Figure 02: Gray Image, Blur Image and Binary Image*

* **contour(thresh):** Find contours in the binary image and identify the grid contour based on its area and polygonal shape.
  + **cv2.findContours(img, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE):** This function finds contours in the binary image (img) using the RETR\_TREE retrieval mode and CHAIN\_APPROX\_SIMPLE contour approximation method. The result is a list of contours and a hierarchy.
  + Then find the contour with the maximum area which is the contour of the grid. The code iterates through contours in a binary image, filtering by area (>25000), approximates their polygonal shape, and selects the contour with the largest area and a quadrilateral shape, assuming it represents a Sudoku grid. The identified contour is stored as grid\_contour.

1. **extract\_digit(frame, grid\_contour):** Call necessary functions to extract the Sudoku grid, digit values, and solve the puzzle.

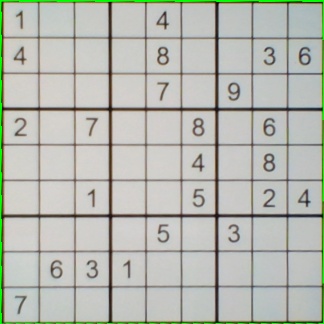
* **cv2.drawContours(frame, [grid\_contour], 0, (0, 255, 0), 2):** Draw the grid contour on the original image

A screenshot of a game

Description automatically generated

*Figure 03: Contour found image*

* **wrap\_perspective(frame, grid\_contour)**: performs perspective transformation on the input image (frame) to obtain a top-down view of the Sudoku grid.
  + Sort Corners:
    - **points = np.vstack(grid\_contour).squeeze():** Extract and flatten the corners of the input grid contour.
    - **points = sorted(points, key=operator.itemgetter(1)):** Sort the flattened corners based on their y-coordinates, ordering from top to bottom.
  + Determine Perspective Transform Points (pts1 and pts2): Check the relative positions of the sorted corners to determine the transformation points (pts1).
    - If the first corner's x-coordinate is less than the second corner's x-coordinate. Determine pts1 based on the relative positions of the other two corners.
    - Else if the second corner's x-coordinate is less than the first corner's x-coordinate. Determine pts1 based on the relative positions of the other two corners.
    - **pts2 = np.float32([[0, 0], [grid\_size, 0], [0, grid\_size], [grid\_size, grid\_size]]):** Define the destination points for the perspective transformation (pts2). This represents a square grid of size grid\_size x grid\_size in the top-down view.
  + **M = cv2.getPerspectiveTransform(pts1, pts2):** Calculate the perspective transformation matrix using the source (pts1) and destination (pts2) points.
  + **grid = cv2.warpPerspective(frame, M, (grid\_size, grid\_size)):** Warp the original image (frame) using the calculated transformation matrix (M) to obtain a top-down view of the Sudoku grid (grid) with dimensions (grid\_size, grid\_size)

****

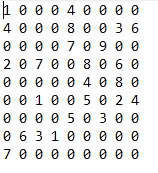
*Figure 04: Top View*

* **get\_cell\_value(grid, sudoku\_grid\_unsolved):** Get the value of each cell in the grid
  + Calculate the boundaries (y2min, y2max, x2min, x2max) for each cell based on its position in the grid.
  + **img = grid[y2min:y2max, x2min:x2max]:** Extract the individual cell from the grid.

  ****  

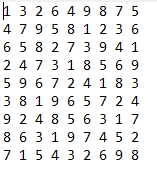
*Figure 05: Divided cells*

* + **img = cv2.resize(img, (28, 28)):** Resize the extracted cell image to a standard size of 28x28 pixels.
  + **img = img.reshape(1, 28, 28, 1):** Reshape the image to fit the expected input shape for the neural network model.
  + **img = img.astype('float32') / 255.0:** Normalize the pixel values to be in the range of [0, 1].
  + Make Predictions:
  + **predict = classifier.predict(img):** Use the pre-trained neural network (classifier) to predict the digit in the current cell.
  + **predicted\_class = np.argmax(predict[0]):** Identify the class (digit) with the highest probability among the predicted classes.
  + **row.append(predicted\_class):** Add the predicted digit to the current row.
  + **sudoku\_grid\_unsolved.append(row):** Add the list of digits in the current row to the overall Sudoku grid (sudoku\_grid\_unsolved).
* Save the extracted grid values to a text file and send it to the solver.



*Figure 06: Extracted grid values*

1. **run\_solver(sudoku\_grid\_solved):** Compile C++ sudoku solver and run compiled program with the saved grid text file as the input. If solution found, output it as a grid\_output.txt text file.

****

*Figure 07: Solved puzzel*

1. **overlay\_grid(frame, pts1, pts2, sudoku\_grid\_solved, sudoku\_grid\_unsolved):** Overlay the solved numbers on the original image.

* **copy\_img = np.zeros(shape=(grid\_size, grid\_size, 3), dtype=np.float32)**: Create an empty image with the same dimensions as the top-down view of the Sudoku grid. It has three channels (RGB).
* Iterate through each cell in the solved puzzle array (sudoku\_grid\_solved). If the corresponding cell in the unsolved Sudoku grid (sudoku\_grid\_unsolved) is empty (indicated by 0), calculate the position to overlay the solved number. Use cv2.putText to overlay the solved number on the empty image. The font type is cv2.FONT\_HERSHEY\_SIMPLEX, and color is set to red (0, 0, 255).



*Figure 08: Solved values*

* **M = cv2.getPerspectiveTransform(pts2, pts1):** Calculate the transformation matrix (M) to reverse the perspective transformation applied earlier.
* **copy\_img\_p = cv2.warpPerspective(fond, M, (w, h)):** Apply the reverse perspective transformation to bring the overlay image back to the original perspective.
* Create Mask and Apply Overlay:
  + Convert the overlay image to grayscale (img2gray).
  + Threshold the grayscale image to create a binary mask (mask).
  + Invert the mask (mask\_inv) and convert it to the appropriate type.
  + Use cv2.bitwise\_and to extract the background of the original frame (img1\_bg).
  + Use cv2.bitwise\_and to extract the foreground of the overlay image (img2\_fg).
  + Combine the background and foreground using cv2.add to get the final overlay image (dst).
  + Display the overlay image (dst) using cv2.imshow.



*Figure 09: Output*

# Optimization techniques

# Challenges Faced

# Limitations

* Fixed Grid Sizes: Limited to 9x9 and 16x16 grids, lacks flexibility for other sizes.
* Dependency on Pre-trained Models: Recognition relies heavily on pre-trained models, limiting adaptability.
* Single-Digit Recognition: Primarily designed for single-digit recognition, may struggle with variations.
* Noisy Image Handling: Performance may degrade with noisy or distorted input images.
* Limited Input Sources: Currently accepts images from files or cameras, expansion options limited.
* Dancing Links Algorithm Complexity: Efficient but may face challenges with extremely large grids or real-time requirements.
* User Interaction Challenges: Primarily file-based input, less interactive for users.
* Algorithmic Rigidity: Static pipeline design, potential for enhancement with dynamic processing.

# Future Improvements

In the pursuit of advancing the Sudoku solving algorithm employing computer vision and machine learning, several promising avenues for improvement exist. Firstly, there is a focus on refining digit recognition accuracy within the Sudoku grid by fine-tuning the neural network model and exploring advanced techniques such as ensemble learning. Additionally, efforts can be directed towards making the algorithm more flexible, enabling it to dynamically handle Sudoku puzzles of various sizes and irregular shapes. Real-time processing capabilities, especially for capturing and solving puzzles directly from a camera feed, can be pursued for enhanced user interaction. User interface improvements, accessibility features, and parallelization strategies aiming to optimize user experience and overall performance can be employed. The algorithm's adaptability is further emphasized through the exploration of adaptive configurations and integration of OpenCV functionalities for enhanced computer vision capabilities.