CO4352

Advanced Algorithms

Mini Project

Sudoku Solver

NAME : ILLANGARATHNA A.N.L.

INDEX NUMBER : 19/ENG/033

REGISTRATION NUMBER : EN93917

SUBMISSION DATE : 01/03/2024

**Introduction**

Sudoku stands out as a mind-bending challenge. This project unravels the mystery behind solving Sudoku puzzles. This code gracefully tackles Sudoku puzzles of diverse sizes, intelligently adapting to 9x9 and 16x16 grids. With its adept use of advanced algorithms and dynamic sizing. Promising an efficient and user-friendly solution if exists. This approach works with constraint propagation, recursive backtracking, and dynamic block sizing combined making it a reliable Sudoku solutions generator.

**Methodology**

The Sudoku Puzzle Solver employs a systematic approach to solve complex Sudoku puzzles. The methodology unfolds in several key steps, ensuring an efficient and adaptable solving process:

1. **Puzzle Representation**

* Utilizes a vector of bitmasks to represent each cell in the puzzle.
* Dynamically detects puzzle size by considering the text file, accommodating both 9x9 and 16x16 grids.

1. **Allowed Values Matrix**

* Maintains an 'allowed' matrix, initially permitting all values for each empty cell.
* Dynamically updates allowed values based on filled cells and puzzle constraints.

1. **Recursive Backtracking:**

This is the main algorithm part of the Sudoku Puzzle Solver involves the use of a ***recursive-backtracking algorithm*,** a systematic approach to explore potential solutions. If it is broken down as follows:

**3.1 Recursive Approach:**

* The algorithm takes a recursive approach, it calls itself to solve smaller instances of the puzzle.
* This recursive nature enables the solver to explore different paths until a solution is found or all possibilities are exhausted and done.

**3.2 Exploration of Solutions:**

* The algorithm begins by iterating through each empty cell in the Sudoku puzzle.
* For each empty cell, it attempts to fill it with a valid value based on the puzzle's constraints.

**3.3 Filling Empty Cells:**

* During the iteration, when the algorithm encounters an empty cell, it tries to fill it with a value.
* The algorithm chooses a value and proceeds to the next empty cell, repeating this process until a solution is found.

**3.4 Valid Value Selection:**

* The selection of a valid value is crucial. The algorithm ensures that the chosen value adheres to Sudoku rules, considering row, column, and block constraints.

**3.5 Backtracking Mechanism:**

* If the chosen value leads to a conflict or an unsolvable situation, the algorithm 'backtracks.'
* Backtracking involves undoing the last decision and exploring alternative values for the previous empty cell.

**3.6 Exploration Continues until it finds the solution or fails:**

* The algorithm continues its exploration, trying different values and backtracking when needed, until a valid solution for the entire puzzle is discovered.

**4. Constraint Propagation:**

This code uses concept of constraint propagation which is a mechanism for maintaining the approach to the solution. The target was to enhance the algorithm's efficiency, leading to the exploration of valid solution paths fst manner.

**4.1 Updating the 'Allowed' Matrix:**

* The algorithm keeps track of an 'allowed' matrix, initially allowing all possible values for each empty cell.
* Upon successfully filling a cell, the 'allowed' matrix undergoes updates to reflect the constraints introduced by the filled cell.

**4.2 Restricting Possibilities:**

* After filling a cell, the algorithm identifies the corresponding row, column, and block associated with that cell.
* The 'allowed' matrix is then adjusted to restrict possibilities in these specific areas.

**4.3 Ensuring Valid Solution Paths:**

* Constraint propagation is crucial in ensuring that the algorithm explores only valid solution paths.
* By updating the 'allowed' matrix based on filled cells, the solver avoids pursuing paths that violate Sudoku rules. So the process is towards the result expected.

**4.4 Conflict Avoidance:**

* The constraint propagation helps in conflict avoidance.
* When exploring potential solutions, the algorithm dynamically adjusts the 'allowed' matrix to prevent conflicts within rows, columns, and blocks.

**4.5 Iterative Application:**

* Constraint propagation is an iterative process, occurring each time a cell is filled.
* This iterative application ensures that as the solver progresses, the 'allowed' matrix becomes a more refined representation of valid possibilities for each empty cell.

**4.6 Puzzle Integrity:**

* Constraint propagation contributes significantly to maintaining the integrity of the Sudoku puzzle.
* It aligns with the rules of Sudoku, making certain that the solver adheres to constraints and only pursues solution paths that comply with the puzzle's inherent structure.

1. **Output Generation:**

* Generates console and file-based outputs, providing a clear presentation of the solved puzzle.
* Dynamically handles block size based on puzzle dimensions in the output ass well.

1. **Performance Metrics:**

* Tracks the total number of recursive calls made during the solving process. And prints it.
* Offers insights into algorithm efficiency and complexity.

1. **Program Execution Time:**

* Utilizes the Chrono library to measure overall program execution time.

1. **Dynamic Block Size Handling:**

* Dynamically determines block size based on puzzle size, enhancing adaptability ensures seamless handling of puzzles with varying dimensions.

1. **User-Friendly Output:**

* Provides human-readable outputs for user experience.

****

*Figure 01: Original Images*

**Code Walkthrough**

This is what each function does in the code.

**slope\_averaging\_custom(image, lines)**

It carefully examines each detected line by Hough transformation, calculates its slope and intercept, and then categorizes them into left and right lanes. Finally, it averages the slopes for each side.

**coordinates\_maker(image, line\_parameters)**

Given the slope and intercept, it efficiently calculates the coordinates to draw a line on the image.

**find\_edge\_pixels(edge\_detected\_image, center\_x\_position)**

This function starts from the middle, looks left and right, and collects the pixel of where those lane edges might be.

**sobel\_edge\_detection(image)**

This function wields the Sobel operator to reveal the edges in the gray scaled image. It is about detecting drastic intensity changes.

**5. sobel\_edge\_visualization(image)**

This function is about visualization. It does the sobel operation for the visualization purposes.

**6. find\_and\_visualize\_edge\_pixels(image, center\_x\_position)**

This function not only finds the edge pixels derived using Sobel, but also paints them in different colors for left and right lanes. (figure 04)

7. **line\_intersection\_coordinate(lines)**

This function finds that exact point where detected lane lines intercept each other.

**8. generate\_lines(image, lines)**

Given the intersection point, it draws lines on our image connecting that point to the detected lanes. It is about visualization.

**9. process\_area(image)**

It decides which part of the image is important and masks out the rest. This way, the code focuses only on the region where lanes are likely to be found.

**10. hough\_transform(img)**

It consider the edges found using **find\_and\_visualize\_edge\_pixels** function. Then it finds where lines are using the Hough transform custom implementation method.

**11. find\_lines(hough\_space, thetas, diag\_len, threshold=100)**

It filters out the lines that aren't important, leaving only those that have a significant impact on our lane detection function.

**12. detect\_lanes(img)**

This function conducts the entire lane detection operation. It orchestrates Sobel edge detection, interested area processing, Hough transform, and line filtering to create the visual representation of the detected lanes.

**13. video\_generator(input\_folder, output\_folder, name)**

This function combines all the images into a video. Then the detected lanes comes to life.

**Important.**  
Line 69 has a this code like follows,

for y in range(height//2, height): # Start from the middle of the image and go to the bottom

It is related to the Region Of interest. To define where the script should consider when each image is processed. In it the the bottom half of the image is considered. Below I have compared with this bottom half only and without the bottom half only constraint related outputs generated.

* Without bottom half constraint.

A road with a red and blue line

Description automatically generated

*Figure 05: Unnecessary parts are considered in edge detection*

* With bottom half constraint.



*Figure 06: Only the necessary parts are considered in edge detection*

Outputs  
All the outputs are located in the outputs folder in the github repository  
Both the videos are located in the   
**Outputs  
 |-TestVideo01  
 |-06\_Visualized\_Result**

**|-TestVideo02  
 |-06\_Visualized\_Result**  
both folders