Sudoku Solver User Manual

Alice Luce, Michael Richards, Jaden Krekow. Adrian Rodriguez

For Users

Installing and Launching the Application

The application is provided in a zip file containing the executable and the sudoku board database. Simply unzip the file where you would like the application to be. The executable can be run without any further setup.

Generating a Board

To generate a board, first click on the “New Game” button on the bottom of the UI to open the “New Game” window. Select your preferred difficulty level and board generating algorithm.

The difficulty level refers to the number of cells which are initially filled. When set to “Easy”, the board will generate with 50 cells filled in. When set to “Normal”, the board has 35 pre-filled cells. When set to “Hard”, the board comes with 25 cells filled in.

There are two choices for generator algorithms, “Brute-Force” and “Top Down”. The “Brute-Force” generator generates boards by adding cells at random points and preventing conflicting cells from being placed. It continues until it has reached the required number of pre-filled cells then verifies that the board has a single unique solution. This is faster for the “Hard” option because it reaches 25 cells quickly. The “Top Down” selection uses an algorithm that first creates a fully solved board from a random seed, then it recursively removes random cells, making sure the original board is the only possible. This is faster for “Easy” boards because “Easy” is closer to a full board than an empty one.

Solving a Board

To solve the board manually, click each cell and input a digit. You may also input smaller notes into an empty cell (see The Notes System). Hints and pre-filled cells cannot be modified. When the board is full, it will test if it is solved and display what you got right and wrong before allowing you to modify the remaining incorrect cells.

To solve the board automatically, select “Auto-Solve” from the menu at the bottom of the application to open the “Auto-Solve” window. Next, select the algorithm you would like to use to solve the board. Different algorithms have different runtimes and handle different boards better or worse.

To visualize how the algorithm solves the board, select the visualization dropdown menu. You may select “No visualization” to not visualize the solver and just fill the board with correct values. The other options visualize, showing varying amounts of steps.

“Every Step” causes the algorithm to pause and display its visualization at every single step in the algorithm. This is the slowest method because every step pauses operation for a short time so you can see how the board has changed and read the description of the step. This is recommended for users looking for a full understanding of the solver algorithm.

“Every Few Steps” causes the algorithm to pause and display after a small random number of steps. This is useful to see the algorithm in broad strokes or to increase speed as there are fewer pauses. This is recommended for users who are looking for a general idea of how the algorithm fills the board without seeing repetitive solution steps.

“Every Several Steps” has a larger maximum number of steps to skip over before displaying. It is useful for algorithms that operate through several small steps, algorithms operating on difficult boards, or for showing how an algorithm works in the simplest terms. It is recommended for users trying to create visually interesting recordings or analyzing the broadest patterns of the algorithm.

The Notes System

The notes system allows you to add small digits into empty cells to hold mnemonic information. The intended usage is to use it to store the digits you think could be in a cell, allowing you to track them and rule them out. To activate it, select the “Notes” button on the bottom menu and then select an unsolved cell (One that’s not pre-filled or filled from a hint). Type in any digit to add it to the notes. To remove a note, type the digit again. As long as the Notation Mode is active (represented by the “Notes”

button being bright green), any empty cell can be selected to add or remove notes. To disable Notation Mode and fill cells as normal, press the “Notes” button again. You may overwrite any cell containing notes with a digit as per usual

The Database

The database allows for the ability to save and load Sudoku games. Games that are saved, will be saved to a local file called “sudoku_puzzles.db”. Deletion of this file will cause all saved games to be lost. You can save an active game by clicking the save button at the bottom of the GUI interface. You can continue to do other actions, such as generate a new game, and load this previous game at another time by clicking the Load button.

For Developers

The GUI

The GUI is primarily developed using PySide6, a binding for the Qt framework. It is defined and managed primarily in board.py as the Board class with settings.py accounting for any additional windows.The pop-up windows are separate widgets while the rest of the UI elements are components of the board window.

The *Sudoku Solvers* project is designed to be intuitive and user-friendly, featuring a minimal interface prioritizing quick response time, intractability, and clear algorithm visualization. Due to being a modern framework that's flexible and easy to implement we choose PySide6 as our main tool for the GUI. The tool provides a wide range of customizability widgets that allow for user interaction. With the board being able to be changed at the click of a button it is important for widgets to be customized to reflect the current state of the GUI. With the grid on the GUI it needs to be interactable for inputting numbers into cells and be customizable to include highlighting and changing cell states to preserve sudoku integrity. With the Notes functionality each cell needs to be able to create a nested 3x3 grid and have numbers in a predefined layout to improve readability. Users can visually track each cell's notes and toggle between input modes

for enhanced playability. PySide 6 layout management through QHLayout and QVBoxLayout allows for condensing information in an organized and scalable way.

The Cells

The cells are QLineEdits aligned in a grid. Each one can only take digits 1-9 and can only hold 1 digit. When a cell is filled by the board generation or a hint (or by an auto-solver) it is set to read-only to prevent the user from modifying it. This also tags it not to be modified when the UI switches to Notation mode.

The Notes

Pressing the “Notes” button sets Board.note_mode to True to activate Notation Mode. In Notation Mode, the cells are replaced with a widget combining a set of nine QLineEdits, one for each number. The widget takes the event of typing into it and uses that to fill or empty the respective space that should contain that digit.

The Visualizer

The visualizer system is the most complicated component of the UI. When Board.system_solve() is called whenever the user presses the “Solve” button in the Auto-Solve window. The settings for the solver (the type of algorithm and how many steps to display) recorded from the window and passed as parameters. If the number of steps is -1, implying the user does not want to visualize the algorithm, the board is simply replaced with the solved board for the sake of time and simplicity. Otherwise, the visualized version of the algorithm is called given the board, the number of steps, and a callback function defined specifically for the algorithm. These callback functions are designed to take information from the solver (passed in through parameters) and information from the previous display (stored in Board) to determine the next display. This involves passing the board to be displayed as well as highlight mapping and text information about the current step to Board.visualize_grid(), which displays the data.

Board.visualize_grid is very similar to the code to initialize the interactive grid for users. It iterates through the 81 cells of the board, filling them in based on the board given by the solver through the callback function. For each cell, fill it with nothing if given

a 0, a single digit if given that digit, or notes if given a list of digits to place in that cell. Each cell will also be highlighted by having the background color of the cell (or sub cells if the cell contains notes.) based on a highlight mapping generated by the callback function and passed to the function.

Because the callback functions are unique to each algorithm, further discussion of each will be found in the algorithm's respective section. In general, each callback is called by its algorithm and passed the board as the algorithm currently has it, along with additional information dependent on the algorithm. This is used to compare to the current board and generate the highlight mapping for the display as well as any additional information output to the user through the text display below. Each callback returns a number of steps before the next callback should be run based on the settings of the board (visualizing every step, every few steps, or every several steps)

Multithreading

During generation of a new board with any generation method, in order to avoid having the GUI freeze during computation, we are using pyside6's QThread library to utilize multithreading. Pyside6 does not allow any UI updates to happen on worker threads, and are only allowed on the main thread, thus we have the heavily computational task of compiling the users settings options and generating a board based off of those settings in a created worker thread. Once those computations are complete and the board is ready to be shown, the thread is killed and the board is shown on the main thread. This behavior is repeated anytime the generate button on the settings page is pressed.

The Generators

There are two board generating algorithms used in the application. Both are found within generators.py along with an unused algorithm that is between them. The two used algorithms are brute_force_generate() and hybrid_top_down(). Both only take the number of hints required for the final board as parameters.

brute_force_generate() does exactly what it says. It generates a board by randomly selecting an empty cell, then checking which digits can be added without

making the board invalid, a random one of these digits is selected and placed. If the board can no longer be added to before it reaches the required number of hints, the generator backtracks. If the board reaches the required amount of hints, it tests the number of solutions for the board, if there is exactly one valid solution, the board is successfully generated and returned. Otherwise, it backtracks and continues randomizing until a successful board is generated.

hybrid_top_down() solves a board based on a random seed and then recursively randomly removes cells. It backtracks if the removed cell results in a board with more than one solution. If the board backtracks too many times (three backtracks per successful removals) a new board is generated from a new random seed to restart the process. The generated board is returned when a removal results in a board with the required number of cells and exactly one solution.

The Settings

There are settings for both generating and solving the board allowing users to customize their experience. Generating boards follow between selecting two different generators and selecting a difficulty before generating the board. Once each option is selected or left on their default values then it will be displayed on the GUI once the user hits generate.

Alternatively, users may opt to instead load a previously saved board or pre loaded board. This is useful for saving time by bypassing generation time or resuming a prior session. Choosing a loading option will cause it to override other settings when generating a board meaning any prior selected options for the generator or difficulty will not be used when a board has any selected board from the database.

The solution settings work similarly to the generator settings but with choosing which algorithm is used in the solution of the current board. A step-by-step solution option is implemented here to give the user some control over pacing and how transparent the system is with its algorithms. Once the system begins to auto solve the current sudoku board it will briefly remove the buttons for normally engaging with sudoku so as to not cause problems during the solving process. They are replaced with widgets designed for visualizer use only such as pausing, resuming, and ending the

visualizer instantly providing the solution skipping all the steps. Some information about each algorithm will be applied to this layout change, showing underneath the visualization widgets.

Settings.py and the Settings class within it are used as an extension of Board.py's Board class to maintain simple options and act as an interface between UI selections and the backend code of the board.

The Database

The database uses SQLite3 to save puzzles locally in a file called "sudoku_puzzles.db". The Primary Key used for PuzzleID is the raw board including blank cells as "0". The Primary Key is the raw board instead of the solved state since one solved state could be associated with multiple valid Sudoku Boards. The board is saved into the database as a string that concatenates all the values in the order of left to right along a single row, and then adds all the rows together starting from top to bottom. The Database class will automatically initialize the .db file if it is not present and allows for saving, updating, retrieving, and deleting boards.

Additionally, the database allows for the ability to create snapshots of boards in time. This feature is not currently used to visualize the board but can be an alternate method instead of using callback functions. Each Snapshot generated has a SnapshotID and has a PuzzleID that references the associated puzzle in the Puzzle table. The SnapshotID increments by a single integer for each snapshot in order of when snapshots were created. When a puzzle is deleted from the database, all associated Snapshots for the Puzzle are also deleted.

Format of database:

Puzzle:

PuzzleID TEXT PRIMARY KEY
RawBoard TEXT NOT NULL
SolvedBoard TEXT NOT NULL
BoardState TEXT

Snapshot:

SnapshotID INTEGER PRIMARY KEY AUTOINCREMENT

PuzzleID TEXT

BoardState TEXT NOT NULL

FOREIGN KEY (PuzzleID) REFERENCES Puzzle(PuzzleID) ON DELETE CASCADE

Brute-Force Algorithm

The Brute-Force algorithm is handled by brute_force() and brute_force_visualize() in bruteforce.py. This is the most rudimentary and simplistic algorithm in the application. The algorithm simply applies the each digit to the first empty cell (left to right, top to bottom), tests if there exists at least one valid digit for the next cell (considering the row, column, and group constraints), and if there is one, it recursively calls the algorithm with the new board. The algorithm backtracks when the next cell has no valid digits. If a complete and valid board is found, it is returned through the chain of recursion up to the source and to the caller.

brute_force() has an additional parameter “count” which defaults to False. If set to true, whenever a valid solved board is found, the algorithm returns 1. If no valid cells exist, the call returns 0. Otherwise the call returns the sum of the return values of each recursive call. In this way the algorithm effectively counts every possible solution of the board. In current implementations it is only used to test if a board has more than 1 valid solution (making it a poorly constructed board), and therefore it short circuits and returns if the sum of valid calls is > 1.

brute_force_visualize() is identical to brute_force() except that it has a callback to the visualizer system implemented after each digit is attempted. The callback compares the currently displayed board with the new one from the algorithm after the digit is attempted (whether it’s valid or not) and highlights digits added since the previous update in green, cells emptied since the previous update in red, and the first empty cell (the one checked for at least one valid option) in light blue. Each digit attempted is counted in a tally passed as an additional parameter and displayed in the description of the algorithm.

Rules-Based Algorithm

The rules-based algorithm is made up of several functions chaining together to apply the various rules a human Sudoku player may use. These include simple ones like ruling out digits that can only rationally be used in one cell in a group to advanced ones comparing the dependencies of different lines. Not all rules are necessary for all boards and in fact for most easy boards all that is needed is to track what digits are permitted in each cell and then filling in cells as they have exactly one digit and ruling the digit out of the remainder. The algorithm operates by reducing the possible digits in each cell (represented by a set of digits in that cell) until there is only one possibility in each cell, allowing each cell to be solved and therefore the whole board.

The algorithm itself is hosted in two files. rulesbased.py and rulesbasedvisualize.py, this is because the code for each version is very large while still being very similar. Containing both in the same file increases shadowing issues and potential for confusion.

Both variations of the algorithm depends on several helper functions from boardutils.py, which provides a variety of utilities to the rest of the codebase. Most of which are simple enough that they're not worth considering or discussing in depth in the manual. The most important utility functions for this algorithm are the iterator-like functions foreach_cell(), foreach_row(), foreach_group() and their "_modify" versions. These each take a 9x9 array (presumably the board), a function to apply on each subsection of the board, and a list to hold additional parameters passed to each call of the function. This is used for rules that can be applied against each row, column and group. The "modify" variants of the algorithms are named as such to imply that they are unsafe to pass boards in that you don't intend to modify, as opposed to the variants without "_modify" in the name which create a deep copy of the board first before passing it to the main variants.

The core difference between the visualizer and non-visualizer variants of the algorithm is the callback function, which is applied after every rule is attempted. The visualizer tracks which cells were read and considered for each rule and passes them as an additional parameter to make sure they are highlighted. Cells that are read but not modified are highlighted in light blue. Cells that have reduced possibilities are

highlighted in light green. Cells at 1 possibility (or just solved) are highlighted in dark green. Cells filled with a guess through the Nishio rule are yellow and highlighted in red when backtracked.

Both initially fill in the given hints and rule out digits based on the basic constraints of the puzzle then apply the rules repeatedly in a loop until the board is solved (or stopping to use Nishio if the board does not change after a full application of the rules).

The first rule applied is simply filling in the cells given by the initial board using fill_cell(). This simply sets the given cell to a given digit and then iterates through every cell in the same row, column and group. If the digit is within the possibilities of any unsolved cell it will be removed, if this results in a cell being reduced to exactly one possibility, fill_cell() is recursively called on it after all work has been done in this call. This rule and Nishio are the only rules that are not called directly in the main loop. This is only called when another rule creates a cell with exactly one possibility.

The next rule, naked_single() is the simplest rule. It checks each cell to see if it has exactly one possibility and calls fill_cell() on it. This is useful for ruled-out cells that slip through the cracks of other rules through indirect ruling out.

The one following that is only marginally more complicated. hidden_single() searches a row/col/group (called using foreach_row_modify(), foreach_col_modify, and foreach_group_modify() which is given a specific hidden_single_group() variant) for a digit that's used exactly once and calls fill_cell() on it.

The next rule is handled by naked_tuple_line() and naked_tuple_group(), which reformats the cells before passing them to naked_tuple_line(). The rule is called similarly to hidden_single() and hidden_single_group(). It checks if there are exactly as many duplicates of a cell as there are possibilities in that cell (E.G. exactly three cells can only be 1, 2, or 3) then each of those cells must contain one of those digits, and therefore these digits are not possible in any other cell in the section and they can be ruled out.

The next rule is very similar in theory but much more complicated in execution. hidden_tuple() and its group variant seek sets of digits that are held in common by a set of cells with the same size (E.G. exactly three cells contain the digits 1, 2, and 3, and

can contain any other digits as well). This is done by creating a dict that maps each digit to the cells its available in then iteratively searching for increasing size tuples by checking which digits are available in at least N cells for a hidden N-tuple. If there are at least N digits with at least N cells, then their cells are compared to see if exactly N digits have the same set of cells. When this is discovered, all digits in those cells are ruled out.

The next rule, xwing() is the most complex rule. It searches for two parallel lines (row or column doesn't matter, but the two must be the same) that each have a digit possible in exactly two cells which themselves are on the same axis as each other (essentially forming the corners of a rectangle). If this unique situation is found, the digits can be ruled out from all cells in all four lines except for those corners because they depend on each other. When one corner is known, the others will also be known. This is rule is utilized by first searching a given line for a hidden 2-tuple then searching for a parallel line with an identically positioned hidden 2-tuple with the same digits. If all conditions are met, the four lines and four corners are ruled out appropriately.

The final of the repeated rules is claiming which is implemented by claiming_line() and claiming_group(). Both search for digits that are only possible in the overlap of the initial group and the other group type (the line variant searches each set of three cells that overlap with a specific group, the group variant searches each set of three cells that overlaps with a specific row or column) and if there is an exclusive overlap then the digit can be ruled out from either set outside of the overlap since it is depended upon by at least one it can depended upon by both.

The final rule implemented is nishio() which is when a solving human imagines if a cell had a specific digit and checks to see if it leads to a contradiction that can allow the possibility to be ruled out. It is generally used as a last resort. Nishio is effectively implemented as a brute force search. The whole algorithm is called again with the first unsolved cell filled in with its first possibility. If the algorithm finds a contradiction (such as an unsolved cell with no possibilities) then it returns False and the caller rules out the possibility and returns to another loop of the core rules to see if it can solve the board yet. Otherwise it returns the solved board. Ideally each nishio search results in a

substantially reduced search space due to the other rules cascading off of the new information, resulting in very few necessary recursive calls.

Hyper-Arc Consistency Algorithm

The Hyper-arc Consistency Algorithm is a constraint propagation algorithm that creates a flow network between either a row, column, or subgrid at a time. This flow network has a capacity of 1 on each edge. It maps from each cell in the subgroup (row, column, subgrid) to all valid assignments left in the domain (only 1 option if the cell has an assignment already), and then back to a set of numbers (1-9) which are the unique valid numbers for the subgroup. The nature of the structure of the flow graph prevents 2 cells from assigning the same number to a cell since the capacity for the final number nodes are 1, two cells can’t flow into the same final node. Through calculating maximal flows in this network, it calculates possible different flows through the network that would result in all cells finding a unique number. These valid assignments are kept in the domain of the cell, and all others that could not find any combination are pruned out. Since the flow network operates off of the domain available on a cell, and not just the assignment, it can take into account indirect conflicts resulting from another subgroup. If no further pruning can occur through all the values in the board, the program will begin guessing values, and then continue pruning after the guess. If a guess is determined to be invalid, the algorithm will backtrack from that previous incorrect guess.

The create_initial_variables(grid) function initializes the Sudoku grid into a structured format. It takes a 9x9 grid as input, where 0 represents an empty cell, and returns a 9x9 list of dictionaries. Each dictionary contains the cell's position (row, col), its domain of possible values, and its subgrid index (block).

The get_constraint_groups(variables) function groups cells into constraint sets (rows, columns, and subgrids). It processes the output from create_initial_variables and returns a list of lists, where each sublist represents a group of cells sharing a constraint.

The is_consistent_assignment(variables, row, col, value) function checks if assigning a value to a cell at (row, col) violates Sudoku rules. It examines the cell's row,

column, and subgrid for duplicates and returns False if the assignment conflicts with existing values.

The find_minimum_remaining_values_cell(variables) function selects the next cell to assign using the Minimum Remaining Values (MRV) heuristic. It iterates through unsolved cells (domain size > 1) and returns the cell with the fewest remaining values, optimizing the backtracking process.

The save_state(variables) and restore_state(variables, backup) functions manage state preservation for backtracking. The former creates a deep copy of all domains, while the latter reverts the board to a previously saved state.

The update_callback(variables, callback, rule, read_cells, initial_filled_cells) function updates an external visualizer during solving. It converts the variables into a 9x9 board state, and marks unsolved cells as 0. This includes the domain state for each cell to allow for the visualization of the domain being reduced as values are pruned.

The filter_group(group) function enforces hyperarc consistency on a constraint group using maximum flow. It models the problem as a flow network, computes the maximum flow to check feasibility, and reduces domains using residual graphs and strongly connected components. The strongly connected components are values where multiple valid solutions were found with the current domains, and as a result are all possible valid assignments. If a value is not determined to be a strongly connected component, it is removed from the domain.

The enforce_hyper_arc_consistency(constraints, variables, callback, initial_filled_cells) function iteratively applies filter_group until no further domain reductions occur. It stops early if a conflict is detected (empty domain). If a conflict is detected, backtracking needs to occur. If no further domain reductions can occur, the program needs to make a guess as no pruning can occur with available information.

The backtrack_solve(variables, constraints, callback, initial_filled_cells) function recursively solves the puzzle using backtracking and constraint propagation. It selects cells via MRV, tests assignments, propagates constraints, and backtracks if dead-ends are reached. Backtracking and testing assignments is a last resort and will only occur if no pruning is left to take place.

The arc_consistency_solve(board, callback, step) function is the entry point for solving a Sudoku puzzle. It initializes the board, applies constraint propagation, and invokes backtrack_solve. It returns the solved grid or None if no solution exists.

Bee Colony Optimization Algorithm

The bee colony optimization algorithm is a swarm intelligence algorithm that follows the foraging behavior of bees. This is done by creating a population of boards from filling in empty cells randomly from the original board. Once each board in the population is completely filled correctly or not it proceeds to the next step of keeping track of fitness. Fitness is a measurement of validity for each board in the population, it increases if violations to sudoku are detected indicating it doesn't create a solve board or that compared to other boards in fitness it's worse. While each board's fitness is calculated it takes the best fitness even if it's not correct and sets it as the best board (for the time). The board population now attempts to match all cells with issues with cells in the best board. These changes don’t all work but the ones that do help the board's fitness are kept while cells that don't improve the fitness are reverted. It will gradually come closer to a correct answer before finally finding a fitness of 0.

bee_colony_search(board, callback=None, top_boards=3) function calls intialization_pop to generate filled boards for the population. Then it runs all the boards in the population for the set generation count. This loop will handle replacing and storing board populations with lower fitness. It will also stop the algorithm early if fitness ever equals 0. The callback function aims to highlight the best board.

initialization_pop(board, callback=None, stop_flag=None) this function will call other functions to fill empty spaces and mutate it. Once a board for the population is created it is then added to the board population list.

variation(board, callback=None, stop_flag=None) this function prevents stagnation and improves solution finding. Through mutation, a board will have random cells changed. The change is to prevent a board from having the best fitness of anything 1 or above as this would indicate an incorrect sudoku solution. This will be

visualized with colors indicating change such as green if a cell was still empty, red if a filled cell was changed.

fill_empty_spaces(board) simple function to fill a 9x9 array will random numbers between 1 to 9 by iterating through an for loop that goes through each cell and stops at any that contain 0, indicating that its empty. This create a fully filled board that bee colony optimization can use to compare between other boards and be counted for when computing fitness

fitness(board) this function keeps track of all violations and totals them up for each board in the population list. The violations are any rules a specific cell will break with its current number such as duplicates in rows, column, and 3x3 sub grids.

move(board, best_board, callback=None, stop_flag=None) this function will swap values in a cell in the same location as the best board. This will keep the cell if it successfully reduces fitness or revert it to its

is_filled(board) a simple function to confirm a board has no empty cells