Sudoku Solver

- **Student Tareq Abd EL Razik**
- **Prof Felix Voigtlander**
- **Subject DS Lab**

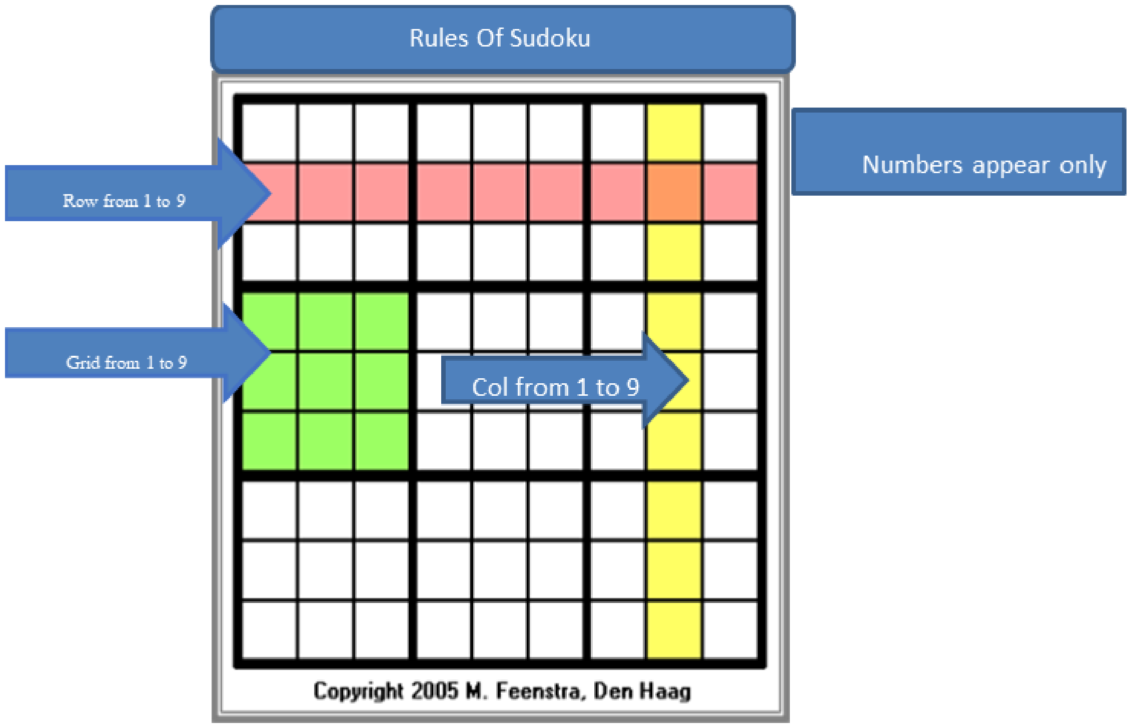
יולי 8

Catholic university Ingolstadt

Introduction

Sudoku GUI Application

This program allows you to play Sudoku puzzles interactively on your computer. Sudoku is a logic-based number placement puzzle game where the objective is to fill a 9x9 grid with digits from 1 to 9. The challenge lies in ensuring that each column and each row and 3x3 sub grid contains all the digits exactly once.



Objective:

- **Interactive Grid:** Play Sudoku on a customizable grid size (9x9 or 16x16).
- **Multiple Difficulty Levels:** Choose from Easy, Medium, or Hard difficulty levels.
- **Save and Load Games:** Save your progress and resume later.
- **Timer and Counter:** Track your progress with a timer and step counter.
- **Solution Solver:** Get help solving puzzles with an automated solver.

Key Components

To achieve these objectives, the project is structured around several key components

- Related Cells:
    - A dictionary that stores the set of cells related to each cell in the Sudoku grid. This includes cells in the same row, column, and sub grid, excluding the cell itself. This structure helps in efficiently applying constraints during the solving process.
- Domains:
    - A dictionary that maps each cell to a set of possible values it can take. Initially, each cell can take any value from 1 to dim. As constraints are applied, these sets are adjusted to reflect the reduced possible values, guiding the solver towards the solution.
- Length Values:
    - A dictionary that tracks the number of possible values remaining for each cell. This helps prioritize which cells to solve next, focusing on those with fewer potential values to expedite the solving process.

Solver's Algorithm

- The solver utilizes constraint propagation techniques to iteratively reduce the possible values for each cell. It employs heuristic search methods, such as the Minimum Remaining Values (MRV) heuristic, to prioritize cells with the fewest possibilities. Backtracking is used to explore potential solutions and revert changes when a contradiction is encountered.

Expected Outcomes

By the end of this project, the Sudoku solver should be able to:

- Solve standard 9x9 and 16x16 Sudoku puzzles efficiently.
- Provide a clear explanation of the solving process.
- Handle puzzles of varying difficulty, demonstrating robustness and flexibility.

algorithm Choice

Constraint satisfaction problems

In the field of programming, especially in the context of constraint satisfaction problems (CSPs), we define the problem with the following components:

- Variables $\boldsymbol{X} = \{\boldsymbol{X1}, \boldsymbol{X2}, \dots, \boldsymbol{Xn}\}$ is finite set of variables.
- Domains A set of domains $\boldsymbol{D} = \{\boldsymbol{D1}, \boldsymbol{D2}, \dots, \boldsymbol{Dn}\}$ where each domain corresponds to the possible values that the variable can take. Formally, the domain is a variable that holds the values and their constraints
- Constraints A set of constraint**s** $\boldsymbol{C} = \{\boldsymbol{C1}, \boldsymbol{C2}, \dots, \boldsymbol{Cn}\}$ that define the rules for the variables. Each is a relation that specifies allowable combinations of values for a subset of variables.

Example:

Definition of Constraint Satisfaction Problems

1. The Set of Values $\boldsymbol{X} = \{\boldsymbol{X1}, \boldsymbol{X2}, \boldsymbol{X3}\}$
2. The set of domains $\boldsymbol{D1} = \{\boldsymbol{1}, \boldsymbol{2}\}$, $\boldsymbol{D2} = \{\boldsymbol{2}, \boldsymbol{3}\}$, $\boldsymbol{D3} = \{\boldsymbol{1}, \boldsymbol{3}\}$
3. The set of constraints $\boldsymbol{C1} = \{\boldsymbol{X1} \mathrel{!=} \boldsymbol{X2}\}$, $\boldsymbol{C2} = \{\boldsymbol{X2} \mathrel{!=} \boldsymbol{X3}\}$

The goal is to find values for $\boldsymbol{X1}$ ***and*** $\boldsymbol{X2}$ ***and*** $\boldsymbol{X3}$ from their respective domains such that all constraints $\boldsymbol{C1}, \boldsymbol{C2}$ are satisfied.

Summary

In this CSP, we are to assign values to variables and for their respective domains while ensuring that $\boldsymbol{X1}, \boldsymbol{X2}, \boldsymbol{X3}$ are not equal to each one another, this method is used to satisfy the sudoku rules by setting constraints for the numbers not to equal each one another we are able to satisfy the sudoku rules.

Algorithm Choice

AC3

The purpose of the AC-3 (Arc Consistency 3) algorithm is to enforce arc consistency in constraint satisfaction problems (CSPs). It iteratively reduces the domains of variables by removing values that do not satisfy the constraints with their neighboring variables. This process simplifies the problem by ensuring that each variable's domain contains only values that are consistent with all related constraints, thereby reducing the overall search space and aiding in finding a solution or determining that no solution exists.

Backtracking

The purpose of the Backtracking algorithm is to systematically explore all possible solutions to a constraint satisfaction problem (CSP) by making choices, exploring consequences, and backtracking when necessary. This approach helps in efficiently finding solutions or determining that no solution exists.

Methodology

Grid Insert

Grid Representation the Sudoku grid is typically represented as an n × n matrix, n ∈ {9, 16}.

$$\boldsymbol{Dim} = \boldsymbol{n} \qquad \boldsymbol{Grid}\ \mathbf{dim} = \sqrt{\boldsymbol{n}} \qquad \boldsymbol{Grird} = \boldsymbol{matrix}$$

Method Assign Grid Properties The initialize grid function prepares the Sudoku puzzle for solving algorithms by getting the puzzle configuration, to be used later by the Solver

Defining Domains Variable

setting cell's domain to all possible values $(\boldsymbol{1}\ \boldsymbol{to}\ \boldsymbol{dim})$, if they are not assigned to a single value.

For cells with preassigned values (non-zero), the domain is restricted to the given value. This prepares the board for constraint propagation in the Sudoku-solving algorithm.

$$f(x) = \begin{cases} \text{if } 1 \le X(i,j) \le n, & \text{then domains(i,j)} : \{X(i,j)\} \\ \text{if } X(i,j) = 0, & \text{then domains(i,j)} : \{1, 2, .., n\} \end{cases}$$

Notation

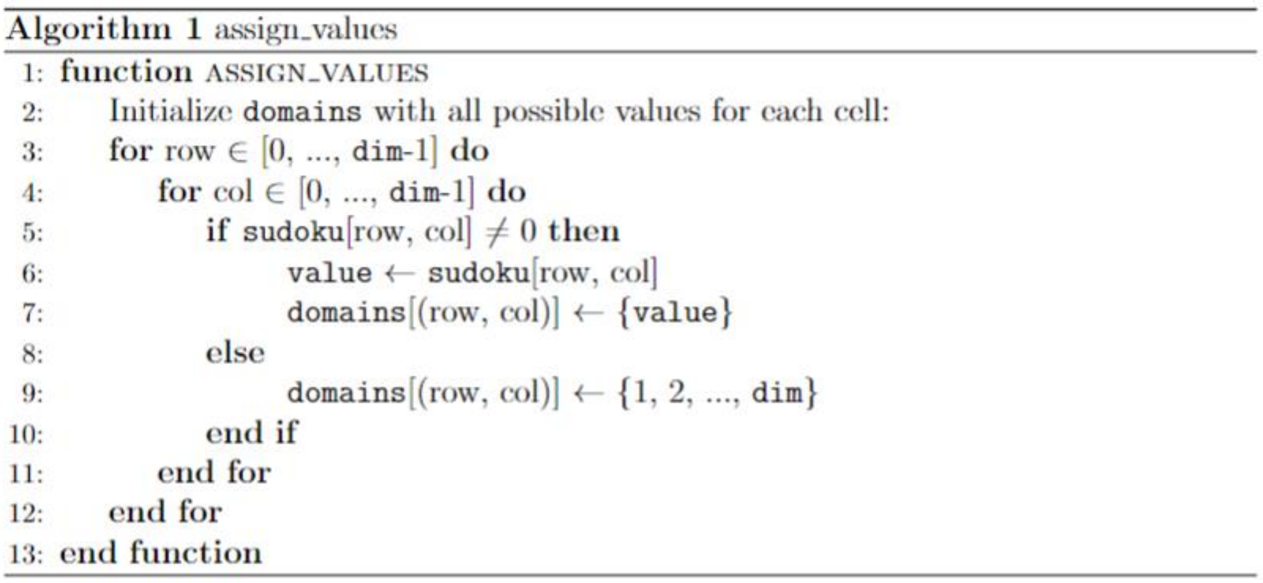
The domains variable is going to hold both the values and the possible values, and its rule is going to be crucial for the solver

Assign values

The assign values function initializes the domains for each cell in the Sudoku board. Each cell's domain is initially set to all possible values (from 1 to dim), except for cells that already have a preassigned value.

Purpose

The purpose of this function is to prepare the initial state of the Sudoku solving process by defining the possible values that each cell can take. This initialization step is crucial for constraint satisfaction algorithms used in solving Sudoku puzzles.



Algorithm 1 assign_values

```
function ASSIGN_VALUES
    Initialize domains with all possible values for each cell:
    for row ∈ [0, ..., dim-1] do
        for col ∈ [0, ..., dim-1] do
            if sudoku[row, col] $\neq$ 0 then
                value ← sudoku[row, col]
                domains[(row, col)] ← {value}
            else
                domains[(row, col)] ← {1, 2, ..., dim}
            end if
        end for
    end for
end function
```

Valid Sudoku

The valid sudoku function verifies whether a given Sudoku board adheres to the rules of Sudoku. A valid Sudoku board should not have any duplicate numbers in any row, column, or 3x3 sub-grid.

Functionality

The function validates Sudoku board correctness by ensuring that:

1) Each row contains unique numbers from 1 to dim.
2) Each column contains unique numbers from 1 to dim.
3) Each 3x3 sub-grid (or box) contains unique numbers from 1 to dim.

Purpose

This function enables us to determine which cells in the Sudoku grid influence each other. It helps assess if a Sudoku puzzle is solvable and guides solving attempts

Related Cells

The aim of the calculate related cell's function is to establish and record the relationships between each cell in a Sudoku grid and its related cells. Specifically:

How It Works

1. Identifying Related Cells:
    - The function identifies all cells related to each target cell. These related cells are located in the same row, column, or sub grid as the target cell.
2. Iteration:
    - The function iterates through each cell in the grid, systematically checking all rows and columns, as well as the sub grid that contains the target cell.
3. Storing Related Cells:
    - Once the related cells for a target cell are identified, they are stored in a dictionary. This allows for constant time access to the related cells, significantly improving the algorithm's efficiency.

Benefits

- Efficiency:
    - By precomputing and storing the related cells in a dictionary, the function ensures that related cells can be accessed in constant time during the solving process. This reduces computational overhead.
- Reduced Computational Space:
    - The dictionary structure enables efficient storage and retrieval of related cells, thereby minimizing the computational space required for solving the Sudoku puzzle.

Algorithm 1 Calculate Related Cells

```
Input: Dimension of Sudoku grid $dim$, Subgrid dimension $grid\_dim$
Output: Related cells for each cell in the grid
Initialize a dictionary $related\_cells$
for each row $r$ from 0 to $dim - 1$ do
    for each column $c$ from 0 to $dim - 1$ do
        $cell\_coords \leftarrow (r, c)$
        $related\_cells[cell\_coords] \leftarrow get\_related\_cells(cell\_coords)$
    end for
end for
return $related\_cells$
```

Algorithm 2 Get Related Cells

```
Input: Coordinates of the cell $cell\_coords$
Output: A set of coordinates of all related cells
Initialize an empty set $related\_cells$
for each index $i$ from 0 to $dim - 1$ do
    Add $(i, cell\_coords[1])$ to $related\_cells$ {Add cells in the same column}
    Add $(cell\_coords[0], i)$ to $related\_cells$ {Add cells in the same row}
end for
$subgrid\_row\_start \leftarrow \left\lfloor \frac{cell\_coords[0]}{grid\_dim} \right\rfloor \times grid\_dim$
$subgrid\_col\_start \leftarrow \left\lfloor \frac{cell\_coords[1]}{grid\_dim} \right\rfloor \times grid\_dim$
for each row $r$ from $subgrid\_row\_start$ to $subgrid\_row\_start + grid\_dim - 1$ do
    for each column $c$ from $subgrid\_col\_start$ to $subgrid\_col\_start + grid\_dim - 1$ do
        Add $(r, c)$ to $related\_cells$ {Add cells in the same subgrid}
    end for
end for
Remove $cell\_coords$ from $related\_cells$ {Remove the original cell from the set of related cells}
return $related\_cells$
```

Purpose

Calculating related cells enables quick iteration over related cells of a target cell, improving efficiency in Sudoku solving and similar tasks.

Sudoku Reduction

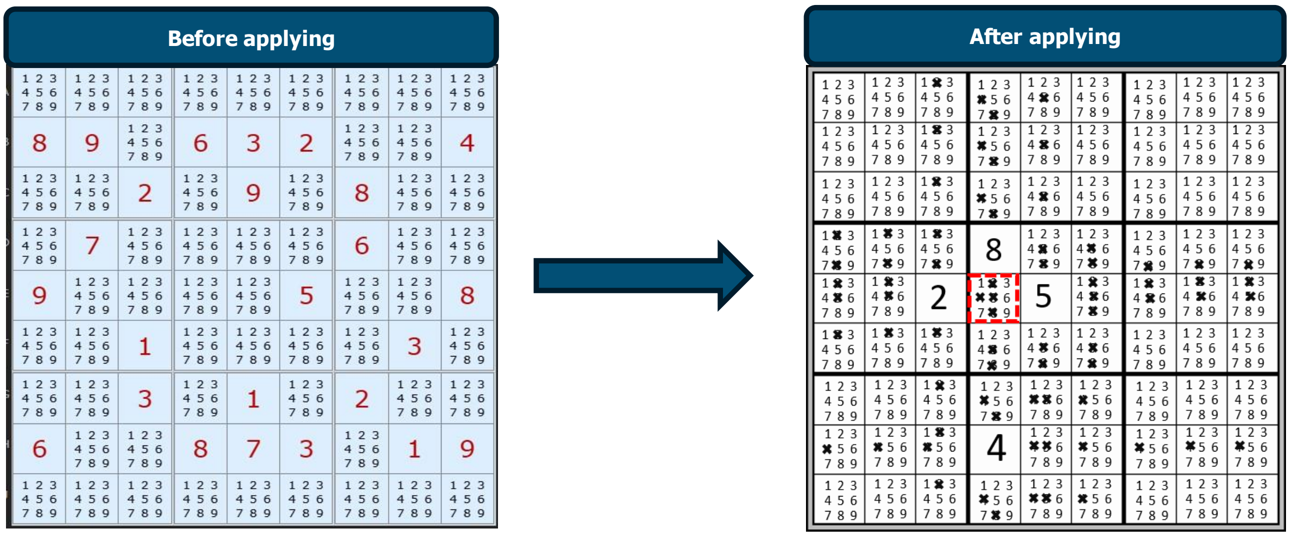
The aim of the sudoku reduction function is to reduce the domain values within a Sudoku grid based on cells that have sing valued domains

How It Works:

1. Identifying single valued Domains
   - Identifies cells with single valued domains (single possible value).
2. Iteration
   - Iterates through each cell to find singleton domains across rows and columns.
3. Updating Related Cells
   - For each cell with a singleton domain, updates related cells' domains (same row, column, or sub grid).

Benefits:

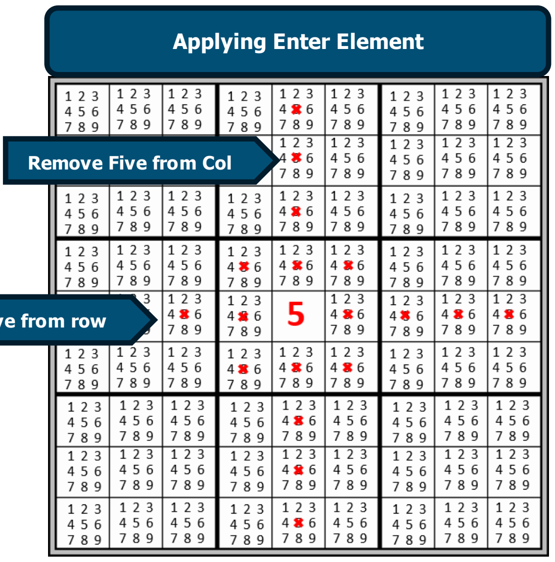
- Efficiency
  - Quickly reduces potential values by targeting cells with singleton domains, optimizing solving process.
- Space Optimization
  - Uses compact singleton domains to minimize memory usage, improving storage efficiency.
- Constraint Propagation
  - Propagates constraints effectively to related cells, guiding subsequent solving steps with clarity.
- Solution Quality Improvement
  - Refines potential values incrementally, enhancing clarity and structure in Sudoku puzzle-solving.
- Algorithm Integration
  - Integral component in Sudoku-solving algorithms, handling diverse puzzle complexities efficiently.
- Practical Application
  - Demonstrates practical utility across Sudoku puzzles, showcasing effective problem-solving strategies.



Enter Element

Enter element machine Function:

- **Assigns** value to Sudoku cell (row, col).
- **Checks** if value is valid in current domain (domains [(row, col)]).
- **Updates** related cells and reduces their domains.
- **Returns** True for success, False if value can't be assigned.



Reduction related cells Function:

- **Reduces** domains of cells related to key after assigning value.
- **Iterates** through related cells, updating constraints.
- **Recursively** reduces related cells' domains, checks Sudoku validity.
- **Returns** True for success, False if Sudoku is invalid.

Benefits:

- **Efficient Assignment:** Quickly assigns values and updates related cells.
- **Constraint Propagation:** Minimizes search space by propagating constraints.
- **Maintains Validity:** Ensures Sudoku rules are consistently enforced.

Summary

This function serves as the backbone of the algorithm because it efficiently assigns values to cells while simultaneously reducing the domain values based on constraints. By doing so, it significantly narrows down the search space by eliminating many potential values. This dual action of assigning values and reducing domains ensures that the Sudoku solving process remains focused and efficient, leading to quicker resolution of the puzzle.

Algorithm 1 Enter Element Machine

```
Require: domains: Dictionary mapping cells to their possible values
    remaining_values: Dictionary tracking remaining possible values for each cell
    row: Row index of the cell
    col: Column index of the cell
    value: The value to be assigned to the cell
Ensure: True if the value was successfully assigned and related cells updated,
    False otherwise
if value in domains[(row, col)] then
    domains[(row, col)] ← {value} // Assign the value to the cell's domain
    if (row, col) in remaining_values then
        Remove (row, col) from remaining_values // Cell is now assigned a single value
    end if
    return reduction_related_cells(domains, remaining_values, (row, col), value, related_cells[(row, col)])
else
    return False // Value cannot be assigned to the cell
end if
```

Algorithm 1 Reduction Related Cells

```
Require: domains: Dictionary mapping cells to their possible values
    remaining_values: Dictionary tracking remaining possible values for each cell
    key: Coordinates (row, col) of the cell where a value was assigned
    value: The value assigned to the cell at key
    relation: Set of coordinates of cells related to the cell at key
Ensure: True if reduction is successful for all related cells, False if a contradiction is found
for each index in relation do
    if value in domains[index] then
        Remove value from domains[index]
        Decrement remaining_values[index] by 1
        if size of domains[index] == 1 then
            single_value ← get_single_value(domains[index])
            if is_valid_sudoku(domains, index, single_value) then
                if not reduction_related_cells(domains, remaining_values, index, single_value, related_cells[index]) then
                    return False
                end if
            else
                return False
            end if
        end if
    end if
end for
return True
```

Unfinished Cells

Significance in Sudoku-solving Algorithms

- Optimizing Search Strategies: By prioritizing cells with the fewest remaining possible values, the function directs the solving algorithm to explore fewer complex branches first. This targeted approach minimizes the number of potential solutions to consider at each step, hence optimizing the search process.

- Reducing Computational Complexity: Starting with cells that have fewer options reduces the overall computational load. It narrows down the initial set of branches to explore, making the solving process more efficient and manageable.
- Enhancing Efficiency in Puzzle Solving: The function's implementation enhances the solving process, making each move clearer and more straightforward. This focused approach speeds up the algorithm's ability to find the correct solution by eliminating non promising paths early on.

Advantages

- Clearer Steps: Each move in solving the Sudoku puzzle becomes more transparent and logical as the algorithm selects cells with fewer possible values to handle next.
- Faster Solutions: By starting with a smaller set of potential numbers for each cell, the solver can more efficiently navigate through the puzzle space to find the correct solution.

Algorithm 1 get_unfinished_cell

```
1: function GET_UNFINISHED_CELL(domains, remaining_values)
2:     if len(remaining_values) == 0 then
3:         return None
4:     end if
5:     cell_with_fewest_values ← MIN(remaining_values, key=remaining_values.get)
6:     possible_values ← domains[cell_with_fewest_values]
7:     return (cell_with_fewest_values, possible_values)
8: end function
```

Note

while the get unfinished cell function might seem straightforward, its role in Sudoku-solving algorithms is critical for achieving efficient and effective puzzle solutions, making it a fundamental part of automated Sudoku solving and optimization processes

parallel backtrack

The aim of backtracking is to methodically explore potential solutions to a problem by navigating through all feasible paths available. This approach involves:

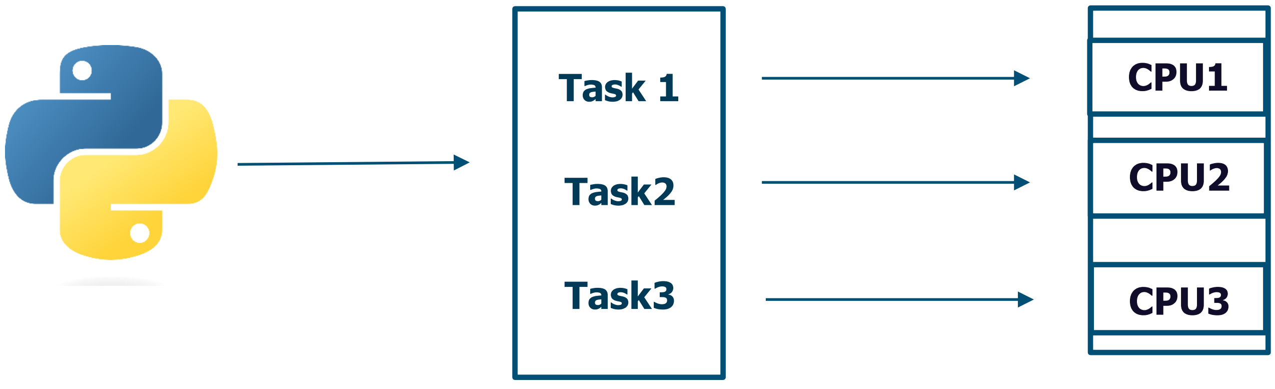
Summary of parallel backtrack Function

The function uses parallel processing to solve a Sudoku puzzle or similar problem:

- It identifies an unfinished cell in the Sudoku grid.
- If all cells are assigned (cell is None), it returns the solved grid (domains).
- Utilizes multiprocessing to handle multiple potential values for a cell at once.
- Maps the different values in possible values into the function (enter in parallel)
- Returns either the solved Sudoku grid (domains) or False if no solution is found.

This approach enhances efficiency by exploring multiple paths simultaneously through parallel backtracking, leveraging the available CPU cores for faster problem solving.

Note The parallelization process operates as follows



Algorithm 1 parallel_backtrack(domains, length_values)

```
1:  function PARALLEL_BACKTRACK($domains, length\_values$)
2:      $cell \leftarrow$ GET_UNFINISHED_CELL($domains, length\_values$)
3:      if $cell$ is None then
4:          return $domains$                                          ▷ Solution found
5:      end if
6:      $(row, col), possible\_values \leftarrow cell[0], cell[1]$
7:      $results \leftarrow list()$
8:      num_cores $\leftarrow$ MULTIPROCESSING.CPU_COUNT(())
9:      $pool \leftarrow$ NUM OF CPU($num\_cores$)
10:     for $value$ in $possible\_values$ do
11:         $result \leftarrow$ POOL.APPLY_ASYNC(function enter_in_parallel, args=(row, col,
    value, domains, length_values))
12:         RESULTS.APPEND($result$)
13:     end for
14:     return PROCESS_RESULTS($results, pool$)
15: end function
```

Solver

The Solver function is designed to solve a Sudoku puzzle using a parallel backtracking approach after performing initial validation and reduction steps. Here's a breakdown of its functionality:

1. **Validation Check**
   - The function begins by checking if the Sudoku puzzle is valid (Not Valid Sudoku condition). If the puzzle is determined to be invalid, the function immediately returns False, indicating that it cannot proceed with solving the puzzle.
2. **Sudoku Reduction**
   - It is called a function (Sudoku Reduction function) that likely performs initial reduction techniques on the Sudoku puzzle. These techniques typically involve eliminating impossible values from cells based on the rules of Sudoku.
3. **Initialization**
   - Initializes solved sudoku as an empty list. This list will eventually hold the solved Sudoku puzzle.
4. **Parallel Backtracking**
   - Executes a parallel backtracking algorithm to solve the Sudoku puzzle. Backtracking is a recursive search algorithm commonly used for solving constraint satisfaction problems like Sudoku. Parallelism here suggests that the algorithm might use multiple threads or processes to explore different paths simultaneously, enhancing efficiency.
5. **Return Result**
   - Finally, the function returns the result of the parallel backtracking algorithm. This result is expected to be the solved Sudoku puzzle, represented possibly as a complete set of values for each cell

Summary

The Solver function encapsulates the entire process of solving a Sudoku puzzle within a software application or algorithmic framework. It integrates initial validation, puzzle reduction, and advanced solving techniques (parallel backtracking) to achieve an efficient and accurate solution to the puzzle.

Algorithm 1 Solver()

```
function SOLVER
    if Not Valid Sudoku then
        return False
    end if
    Call the Sudoku Reduction function.                ▷ removing invalid values
    self.solved_sudoku = []                             ▷ Store solved puzzles
                                                    ▷ parallel backtracking to solve
     return $self.parallel\_backtrack(self.domains, self.length\_values)$
end function
```

process results

The aim of the process results function is to iterate through the collection of outputs from assigned tasks stored in results and continues to wait until the desired number of solutions (max solutions) is achieved

Functionality

  - Manages parallel task completion.
  - Checks task completion (result. Ready ()) and collects valid outputs.
  - Terminates early upon reaching max solutions.
  - Includes a timeout (time. Sleep(...)) to prevent deadlock.
- Return Value:
  - Returns True if max solutions are found.
  - Returns False if the desired solutions are not reached.
- Usage Context:
  - Used in parallel computing to efficiently manage task completion and solution collection.
  - Prevents potential deadlock scenarios with timeout mechanism.

This function efficiently handles parallel tasks, ensuring controlled aggregation of results and early termination when sufficient solutions are found.

Algorithm 1 Process Results

```
function PROCESS_RESULTS(results, pool, solutions = None, max_solutions = 1)
    while results not empty do
        ready ← False
        for each result in results do                                ▷ Iterate
            if result is ready then
                ready ← True
                output ← get result
                if output not None then
                    append output to solved_sudoku
                end if
                remove result from results
                if length of solved_sudoku ≥ max_solutions then
                    close pool
                    return True                       ▷ Exit if enough solutions
                end if
            end if
        end for
        if ¬ready then
            sleep to prevent deadlock
        end if
    end while
    close pool
    return False                          ▷ Exit if max_solutions not reached
end function
```

try value

aims to solve a Sudoku puzzle using recursive backtracking, systematically exploring all possible solutions by traversing each branch until it successfully solves the Sudoku grid.

- Process:
    1. Base Case Check:
        - Checks if all cells are assigned (cell is None). If true, returns domains, indicating the Sudoku is solved.
    2. State Backup:
        - Creates backups (domains backup) of the current state of domains using dictionary comprehension.
    3. Iteration Over Possible Values:
        - Iterates through possible values (possible values) for the current cell.
    4. Value Assignment:
        - Tries assigning each value using enter element machine.
    5. Recursive Call:
        - Recursively calls (try value) with the updated domains and (length values) after attempting to assign a value.
        - If a solution (result) is found in the recursive call, returns the solution immediately.
    6. State Restoration:
        - Restores domains to their previous state (domains backup) before trying the next value.
    7. Return False:
        - If no valid solution is found after trying all possible values (for value in possible values), returns False.

Usage

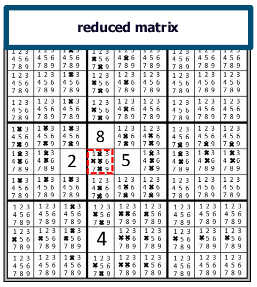
The try value function efficiently explores different value assignments for Sudoku cells using recursive backtracking. It ensures integrity by backing up and restoring the state of domains during value assignments and backtracking. The function systematically attempts to solve the Sudoku puzzle by exploring potential solutions until either success or exhaustion of possibilities, returning the solved Sudoku grid (domains) or False if no solution exists.

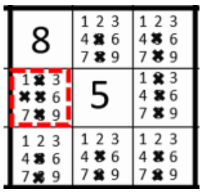
Algorithm 1 Try Value Method

```
function TRY_VALUE($domains$, $length\_values$)
    $cell \leftarrow$ get_unfinished_cell(domains, length_values)
    if $cell$ is None then
        return $domains$                         ▷ Return solved Sudoku grid if all cells are assigned
    end if
    $(row, col), possible\_values \leftarrow cell[0], cell[1]$
    $domains\_backup \leftarrow \{k : v.\text{copy}() \text{ for } k, v \text{ in } domains.items()\}$
    $length\_values\_backup \leftarrow \{k : v \text{ for } k, v \text{ in } length\_values.items()\}$
    for $value$ in $possible\_values$ do
        if ENTER_ELEMENT_MACHINE($domains$, $length\_values$, $row$, $col$, $value$) then
            $result \leftarrow$ TRY_VALUE($domains$, $length\_values$)
            if $result$ then
                return $result$                  ▷ Return solved Sudoku grid if solution found
            end if
        end if
        $domains \leftarrow \{k : v.\text{copy}() \text{ for } k, v \text{ in } domains\_backup.items()\}$
        $length\_values \leftarrow \{k : v \text{ for } k, v \text{ in } length\_values\_backup.items()\}$
    end for
    return False                                 ▷ Return False if no valid solution found
end function
```

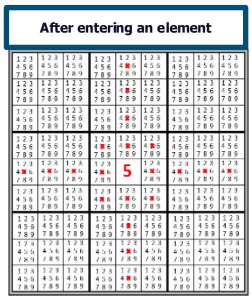
Solver AC3 & Parallel Backtracking

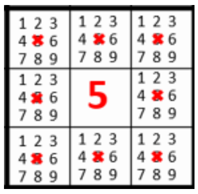
The back tracking algorithm works by first taking the reduced matrix, and then it tries to enter numbers using the available numbers according to the constraints defined by the reduction of the sudoku.



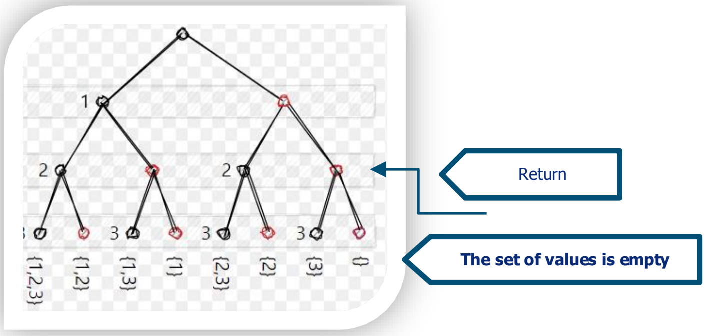


then using the enter element function & sudoku reduction when an element has been entered, we reduce domains according to that element that have been entered

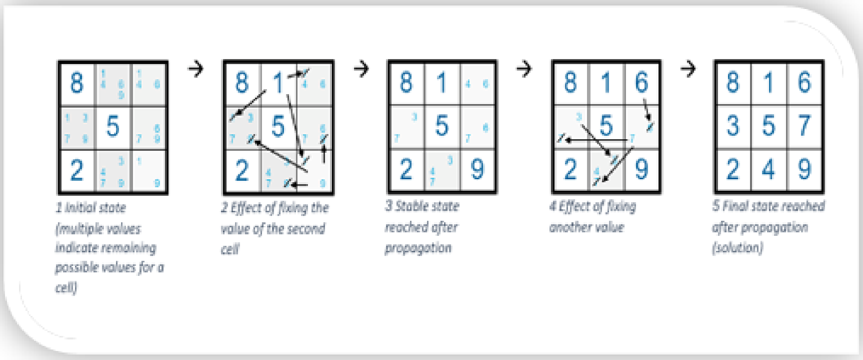




If the entered value does not lead to a suitable solution, the algorithm backtracks. Using backups of the domains and length values, it reverts to the previous state and continues trying other possible values from the list, ensuring all potential solutions are explored system atically.



Example: Backtracking Algorithm with AC3 for Sudoku Solving

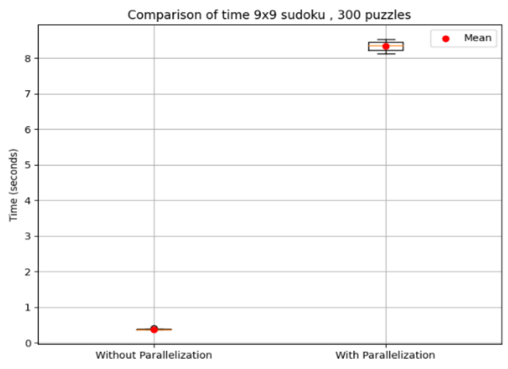

1 Initial state (multiple values indicate remaining possible values for a cell)
2 Effect of fixing the value of the second cell
3 Stable state reached after propagation
4 Effect of fixing another value
5 Final state reached after propagation (solution)

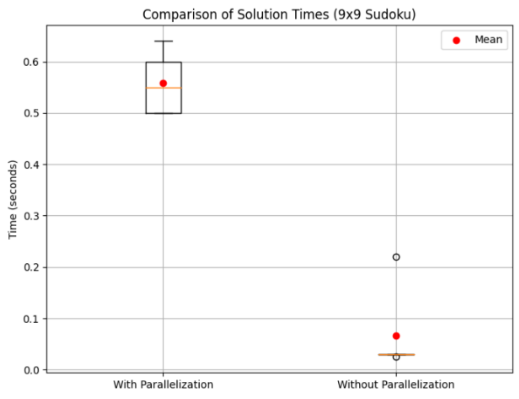
Sudoku Solving Process

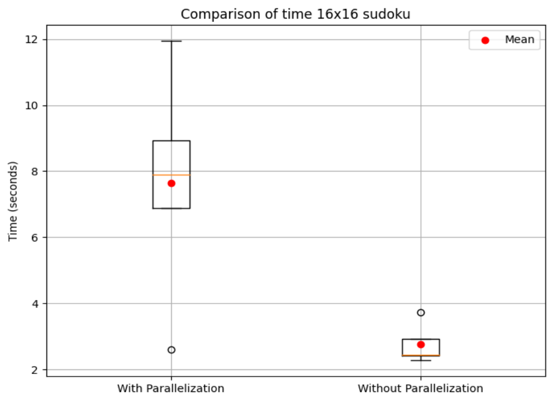
1. Initial Value Assignment:
    - **Assign Initial Values:** Start by placing the given numbers into their respective cells using the initial **assignment function. This sets up the Sudoku grid with the provided values.**
2. Grid Validation:
    - **Validate Initial Grid:** Use the valid sudoku function to check the initial grid configuration. This ensures there are no conflicts, and the puzzle setup is correct and solvable.
3. Domain Reduction: (The AC3 part)
    - **Simplify Domains:** Apply the sudoku reduction function to reduce the domains to their possible values according to the constraints. This step iterates through cells with single possible values and updates the domains of related cells, simplifying the puzzle.
4. Parallel Backtracking Preparation:
    - **Prepare for Solving:** Initialize an empty list to store potential solutions. Set up parallel processing to explore multiple solution paths simultaneously, leveraging available CPU cores for efficiency.
5. Parallel Backtracking Execution:
    - **Identify Unfinished Cells:** Use the parallel backtrack function to locate unsolved cells.
    - **Parallel Value Assignment:** Employ the enter in parallel function to try different values for each unsolved cell using parallel processing. This speeds up the solving process by exploring multiple paths concurrently.
6. Processing Parallel Results:
    - **Collect Outputs:** Manage and collect outputs from parallel tasks using the process results function. It monitors task completion, collects valid outputs, and halts further processing once the desired number of solutions is found. A timeout mechanism is included to avoid deadlocks.
7. Recursive Backtracking (if needed):
    - **Fallback Strategy:** If parallel processing does not yield a solution, revert to the try value function for recursive backtracking. This method systematically explores all possible values for unsolved cells, restoring previous states if an attempt fails.
8. Sudoku Matrix Creation:
    - **Generate 2D Representation:** Use the create sudoku matrix function to create a 2D NumPy array representing the Sudoku grid. This facilitates visualization and verification of the solution.

Results

Based on our observations, it appears that parallelization is negatively impacting performance rather than improving it. The data shows that Sudoku puzzles are solved more slowly with parallelization than without it. This suggests that the added time required to manage and coordinate parallel tasks is greater than the time saved by distributing the workload. As a result, instead of speeding up the solving process, parallelization is causing delays and inefficiencies.







The non-parallelized solver works using the same methods but focuses on finding solutions using a single processor. Its efficiency is by its simplicity, particularly in minimizing the number of elements considered in each cell during computation. This approach makes parallelization less critical, despite its potential efficiency, due to the big eliminations for branches.

Discussion

- **Algorithm** Choice**:** The AC3 algorithm plays a crucial role in simplifying Sudoku puzzles by iteratively reducing the domain of variables based on constraints. It ensures that each variable's domain only contains values consistent with all related constraints, thereby reducing the search space.

- **Constraint** Propagation: Techniques such as Sudoku reduction are pivotal in swiftly narrowing down potential solutions. By identifying and updating cells with singleton domains (cells with only one possible value), this method efficiently propagates constraints across the grid, guiding subsequent solving steps.

- Backtracking Method: Backtracking is employed to systematically explore different number placements, backtrack upon encountering contradictions, and try alternative values until a solution is found. This methodical approach ensures that all potential solutions are explored without unnecessary repetition.

- Efficiency with Parallelism: Parallel backtracking enhances efficiency by utilizing multiple processors to explore various solution paths concurrently. This parallel processing reduces the time taken to find a solution, especially in complex puzzles or scenarios where multiple solutions may exist.

- Validation and Verification: Before attempting to solve a Sudoku puzzle, it's essential to validate the initial grid setup to ensure compliance with Sudoku rules. This initial validation step sets the foundation for a solvable puzzle and prevents wasted computation on invalid configurations.

- Applications Beyond Sudoku: The algorithms used in Sudoku solving, particularly constraint satisfaction algorithms like AC3 and backtracking, find applications in various fields such as scheduling, resource allocation, and configuration problems. Their versatility lies in their ability to handle diverse constraints and optimization objectives.

- Challenges and Future Directions: Challenges include scaling these algorithms to larger grids or more complex rule sets efficiently. Future research may focus on optimizing these algorithms further, potentially integrating machine learning techniques or heuristic improvements for faster and more scalable solutions.

conclusion

Our project represents a significant achievement in integrating a graphical user interface with the AC3 algorithm for solving Sudoku puzzles. Here are the key points highlighted by our work:

1. Enhanced User Experience: The Sudoku GUI offers a seamless interactive experience, allowing users to play on grids of different sizes (9x9 or 16x16) and select from varying difficulty levels (Easy, Medium, Hard).
2. Algorithmic Power of AC3: By incorporating the AC3 algorithm, our application efficiently reduces the puzzle-solving space through constraint propagation. This ensures that each step in solving the Sudoku adheres to the game's logical rules.
3. Versatility and Accessibility: Our project caters to a wide range of users, from casual players to Sudoku enthusiasts, by providing features like saving games, tracking progress with timers, and offering a solution solver for assistance.
4. Practical Application of CSP: The project demonstrates the practical application of Constraint Satisfaction Problems (CSPs) in a gaming context. It showcases how algorithms like AC3 can be used to automate complex problem-solving tasks effectively.
5. Future Directions: Looking ahead, there is potential to further optimize the algorithm, explore additional heuristic methods, and expand the application to handle more challenging Sudoku variants or other puzzle types.
6. Educational Value: Beyond entertainment, our Sudoku solver and GUI serve as educational tools, illustrating fundamental concepts in computer science such as constraint propagation, heuristic search, and graphical user interface design.

By achieving these milestones, our project not only enhances the Sudoku playing experience but also contributes to the broader field of computational problem-solving and software development. It exemplifies the synergy between theoretical algorithms and practical applications in modern software engineering.

References

Understanding Constraint Satisfaction Problems (CSPs) and Their Solution Strategies

https://www.boristhebrave.com/2021/08/30/arc-consistency-explained/

Algorithm to Solve Sudoku

https://www.geeksforgeeks.org/sudoku-backtracking-7/

Solving Sudoku in Elixir with AC-3
https://www.youtube.com/watch?v=ao1CO8_V5do

Implementation of Arc Consistency
https://cse.unl.edu/~choueiry/F12-421-821/Homework/homework2.pdf