EAIRN - Laboratory 2

Variant 4 – SUDOKU solver using Backtracking and Forward checking

Lab Group - 105

Group 9

By Martyna Wielgopolan and Aditya Kandula

Introduction:

This task (Variant 4) involves solving a **Sudoku puzzle** using a **Constraint Satisfaction Problem (CSP)** approach with **backtracking and forward checking**. The puzzle is modeled as a 9x9 grid where empty cells (0) must be filled with digits 1–9, ensuring each row, column, and 3x3 box contains unique values.

The solution uses **backtracking** to explore possible assignments and **forward checking** to eliminate invalid choices early, improving efficiency. A **step-by-step visualization** shows how values are assigned during solving.

This report explains the approach, the role of variables, domains, and constraints, and evaluates the benefits of forward checking through selected test cases.

The algorithm used is **backtracking with forward checking**, a common technique for solving **Constraint Satisfaction Problems (CSPs)** like Sudoku.

Backtracking

Backtracking is a depth-first search algorithm that assigns values to variables one by one. If a conflict arises, it backtracks to try a different value. It ensures all constraints (row, column, and 3x3 box rules) are respected.

Forward Checking

Forward checking improves efficiency by eliminating values from the domains of future variables that would cause conflicts, thus reducing the search space and avoiding dead ends early.

Advantages:

- Guarantees a correct solution if one exists.
- Forward checking speeds up solving by avoiding unnecessary recursion.
- Easy to implement and visualize.

Disadvantages:

- May still be slow for very complex or minimally filled puzzles.
- No guarantee of finding the most optimal path—only the first valid one.

Implementation:

The solution models for Sudoku contains:

```
def __init__(self, variables, domains, constraints):
    self.variables = variables
    self.domains = domains
    self.constraints = constraints
    self.solution = None
    self.viz = []
```

This initializes(constructor which auto run on class implementation) the CSP:

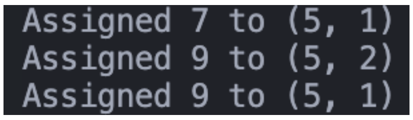
- `variables`: All empty cells in the Sudoku grid (where value is 0)
- `domains`: Possible values (1-9) for each empty cell
- `constraints`: Relationships between cells
- viz: An empty list to track solution steps for visualization

```
def print_sudoku(self, puzzle):
    for i in range(9):
        if i % 3 == 0 and i != 0:
            print("- - - - - - - - - - -")
        row_output = ""
        for j in range(9):
            if j % 3 == 0 and j != 0:
                row_output += " |"
            row_output += f" {puzzle[i][j]}"
        print(row_output.strip())
```

This part of the code displays the SUDOKU table in the terminal for better visualisation.

```
def visualize(self):
    print("\nStep-by-step visualization:")
    for step in self.viz:
        print(f"Assigned {step[2]} to ({step[0]}, {step[1]})")
```

The function `visualize(self)` is used to track each step followed by the programme


```
Assigned 7 to (5, 1)
Assigned 9 to (5, 2)
Assigned 9 to (5, 1)
```

```
    def forward_checking(self, var, value, assignment):
        temp_domains = copy.deepcopy(self.domains)
```

This creates a copy of the domains for all unassigned variables

```
for peer in self.constraints[var]:
            if peer in temp_domains and value in temp_domains[peer]:
                temp_domains[peer].remove(value)
                if not temp_domains[peer]:
                    return None
        return temp_domains
```

- For each "peer" (cells in the same row, column, or 3x3 box)
- If that peer is unassigned and could potentially take the same value
- We remove that value from its domain
- After removing a value, if any domain becomes empty. We immediately return `None` to signal failure.

```
def is_consistent(self, var, value, assignment):
        for neighbor in self.constraints[var]:
            if neighbor in assignment and assignment[neighbor] == value:
                return False
        return True
```

This checks if a value can be legally placed in a cell:

- It looks at all peer cells in the constraints
- If any already assigned peer has the same value, return False
- Otherwise, return True (the assignment is valid)

```
def select_unassigned_variable(self, assignment):
        for var in self.variables:
            if var not in assignment:
                return var
        return None
```

This selects the next unassigned variable (empty cell). The implementation here is basic it just takes the first unassigned variable it finds.

```
def backtrack(self, assignment):
        if len(assignment) == len(self.variables):
            return assignment
        var = self.select_unassigned_variable(assignment)
        for value in self.domains[var]:
            if self.is_consistent(var, value, assignment):
                assignment[var] = value
                self.viz.append((var[0], var[1], value))
                temp_domains = self.forward_checking(var, value, assignment)
                if temp_domains is not None:
                    old_domains = self.domains
                    self.domains = temp_domains
                    result = self.backtrack(assignment)
                    if result:
                        return result
                    self.domains = old_domains
                del assignment[var]
        return None
```

This is the core recursive algorithm:

- If all variables are assigned, we've found a solution
- Otherwise, select an unassigned variable
- Try each possible value in its domain
- For each value:
    - Check if it's consistent with current assignments
    - Add it to the assignment and record the step for visualization
    - Apply forward checking to update domains
    - Recursively try to complete the rest of the puzzle
    - If successful, return the solution
    - If not, backtrack (undo the assignment and try another value)
- If no values work, return None (puzzle is unsolvable from this state)

```
def is_valid_puzzle(self, puzzle):
        def has_duplicates(block):
            nums = [num for num in block if num != 0]
            return len(nums) != len(set(nums))
```

```
    # Check rows
    for row in puzzle:
        if has_duplicates(row):
            return False

    # Check columns
    for col in zip(*puzzle):
        if has_duplicates(col):
            return False

    # Check 3x3 boxes
    for box_x in range(0, 9, 3):
        for box_y in range(0, 9, 3):
            box = [puzzle[i][j] for i in range(box_x, box_x+3) for j in range(box_y, box_y+3)]
            if has_duplicates(box):
                return False

    return True
```

The is_valid_puzzle method is crucial because it ensures the initial Sudoku puzzle doesn't already violate the basic rules of the game. Sudoku requires that each number from 1 to 9 appears only once in every row, column, and 3×3 subgrid. If the input puzzle contains duplicate values in any of these regions, it's unsolvable by definition. By validating the puzzle before attempting to solve it, we avoid wasting computational effort on an invalid configuration and provide immediate feedback to the user.

```
puzzle = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
          [0, 0, 0, 1, 0, 5, 0, 0, 0],
          [0, 9, 8, 0, 0, 0, 0, 6, 0],
          [0, 0, 0, 0, 0, 3, 0, 0, 1],
          [0, 0, 0, 0, 0, 0, 0, 0, 6],
          [0, 0, 0, 0, 0, 0, 2, 8, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 8],
          [0, 0, 0, 0, 0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0, 0, 4, 0, 0]]

variables = [(i, j) for i in range(9) for j in range(9) if puzzle[i][j] == 0]
domains = {(i, j): list(range(1, 10)) for i, j in variables}
```

```
constraints = {}
```

Here we:

- Define the initial puzzle (0s represent empty cells)
- Create variables for each empty cell
- Set the domain of each variable to be all possible values (1-9)

```
for i in range(9):
    for j in range(9):
        if puzzle[i][j] == 0:
            peers = set()
            # row and column
            peers.update(((i, y) for y in range(9) if y != j))
            peers.update(((x, j) for x in range(9) if x != i))
            # 3x3 box
            box_x, box_y = 3 * (i // 3), 3 * (j // 3)
            for x in range(box_x, box_x + 3):
                for y in range(box_y, box_y + 3):
                    if (x, y) != (i, j):
                        peers.add((x, y))
            constraints[(i, j)] = peers
```

This builds the constraints for each empty cell by:

- Finding all cells in the same row
- Finding all cells in the same column
- Finding all cells in the same 3x3 box
- Combining these into a set of peer cells (constraints)

```
csp = CSP(variables, domains, constraints)
sol = csp.solve()

solution = [row[:] for row in puzzle]
if sol:
    for (i, j), val in sol.items():
        solution[i][j] = val

    csp.print_sudoku(solution)
    csp.visualize()
else:
    print("Solution does not exist")
```

Finally:

- If a solution is found, we update the original puzzle with the solution values
- Print the completed Sudoku grid
- Visualize the step-by-step solution process
- If no solution exists, we print a message

Discussion:

Forward checking is implemented to enhance backtracking by pruning the domain of neighboring variables whenever a value is assigned. If assigning a value to a variable causes any of its neighbors to lose all their valid options, the algorithm backtracks early.

Improvement:

- Reduces the number of recursive calls significantly.
- Helps avoid exploring branches that are guaranteed to fail.

Test Cases and Corner Cases:

We tested the algorithm using different Sudoku configurations:

1. **Moderately Difficult Puzzle** (the main test case)
   - Chosen because it contains a typical number of empty cells.
   - Demonstrates correct variable assignment and forward checking in action.

```
puzzle = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
          [0, 0, 0, 1, 0, 5, 0, 0, 0],
          [0, 9, 8, 0, 0, 0, 0, 6, 0],
          [0, 0, 0, 0, 0, 3, 0, 0, 1],
          [0, 0, 0, 0, 0, 0, 0, 0, 6],
          [0, 0, 0, 0, 0, 0, 2, 8, 0],
          [0, 0, 0, 0, 6, 0, 0, 0, 8],
          [0, 8, 0, 0, 0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0, 0, 4, 0, 0]]
```

2. **Nearly Complete Puzzle**
    - Only a few cells are empty.
    - Used to test if the algorithm finishes quickly and correctly in simple cases.

```
puzzle = [[5, 3, 4, 6, 7, 8, 9, 1, 2],
  [6, 7, 2, 1, 9, 5, 3, 4, 8],
  [1, 9, 8, 3, 4, 2, 5, 6, 7],
  [8, 5, 9, 7, 6, 1, 4, 2, 3],
  [4, 2, 6, 8, 5, 3, 7, 9, 1],
  [7, 1, 3, 9, 2, 4, 8, 5, 6],
  [9, 6, 1, 5, 3, 7, 2, 8, 0],  # Only one cell is missing
  [2, 8, 7, 4, 1, 9, 6, 3, 5],
  [3, 4, 5, 2, 8, 6, 1, 7, 9]]
```

3. **Minimal Clue Puzzle (17 clues)**
    - This is the most hardest SUDOKU puzzle known so far.
    - Selected to test performance under heavy branching and deep recursion.

```
puzzle = [[0, 0, 0, 0, 0, 0, 0, 1, 2],
  [0, 0, 0, 0, 3, 5, 0, 0, 0],
  [0, 0, 0, 7, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 3, 0, 0],
  [8, 0, 0, 0, 0, 0, 0, 0, 4],
  [0, 0, 1, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 2, 0, 0, 0],
  [0, 0, 0, 6, 4, 0, 0, 0, 0],
  [9, 2, 0, 0, 0, 0, 0, 0, 0]]
```

4. **Unsolvable Puzzle**
    - A puzzle that violates Sudoku rules or leads to dead ends.

- Useful to check if the algorithm gracefully detects and reports failure.
- And as excepted the algorith stops solving as soon as it realisez that it is not solveable.

```
puzzle = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
      [0, 5, 0, 1, 0, 5, 0, 0, 0], # 2'nd 5 in the first square
      [0, 9, 8, 0, 0, 0, 0, 6, 0],
      [0, 0, 0, 0, 0, 3, 0, 0, 1],
      [0, 0, 0, 0, 0, 0, 0, 0, 6],
      [0, 0, 0, 0, 0, 0, 2, 8, 0],
      [0, 0, 0, 0, 6, 0, 0, 0, 8],
      [0, 8, 0, 0, 0, 0, 0, 1, 0],
      [0, 0, 0, 0, 0, 0, 4, 0, 0]]
```

Each test case was chosen to highlight a different aspect of the algorithm: correctness, performance, or robustness.

Conclusion:

Through this task, we learned how to model a Sudoku puzzle as a **CSP** and solve it using **backtracking with forward checking**. We understood the importance of defining variables, domains, and constraints correctly, and how forward checking can significantly reduce unnecessary computation by pruning invalid paths early.

The most challenging part was managing domain updates during recursion and ensuring correctness when backtracking. Another difficulty was selecting meaningful test cases to evaluate performance under different scenarios.

To improve the solver, we could integrate **heuristics like MRV (Minimum Remaining Values)** to make smarter variable selections and reduce solving time on harder puzzles. Additionally, optimizing the visualization for clarity or adding a graphical interface could make the algorithm more user-friendly and insightful.