Laboratory 2

Variant 4

Class Group 105

Group 24

By Dimitrios Gkolias and Vasileios Ntalas

Introduction

This task involves solving a Sudoku puzzle using a Constraint Satisfaction Problem (CSP) approach. Sudoku is a popular logic-based puzzle where the objective is to fill a 9x9 grid with numbers from 1 to 9, ensuring that each row, column, and 3x3 subgrid contains each number exactly once. This task is a classic example of a combinatorial problem that can be efficiently solved using CSP techniques.

In this approach, we model the Sudoku puzzle as a CSP, where:

- Variables represent the cells of the grid.
- Domains represent the possible values (numbers from 1 to 9) that can be assigned to each variable.
- Constraints enforce the rules of Sudoku: no number can repeat in the same row, column, or 3x3 subgrid.

To solve the CSP, we apply a backtracking algorithm with several heuristics to improve performance. The backtracking algorithm is a depth-first search technique that systematically tries different variable assignments, backtracking when it reaches an invalid state (i.e., when the constraints are violated).

The backtracking algorithm used in this task is a general method for solving CSPs, and it is widely applied to problems like Sudoku, crossword puzzles, and map coloring. It is based on exploring all possible variable assignments and undoing assignments (backtracking) when a dead-end is encountered. The algorithm iterates through the search space recursively, making decisions step-by-step and backtracking when necessary to explore alternative solutions.

Key heuristics employed in this algorithm are:

1. **Minimum Remaining Values (MRV)**: This heuristic selects the variable with the fewest remaining legal values in its domain, which helps to reduce the search space and guides the algorithm to make more promising decisions early on.

2. **Least Constraining Value (LCV)**: Once a variable is selected, this heuristic tries to assign the value that removes the fewest options from the domains of neighboring variables. This heuristic reduces the chances of encountering conflicts later on.
3. **Forward Checking**: This technique ensures that, after assigning a value to a variable, the domains of related variables are updated by removing any conflicting values. If a domain of any related variable becomes empty, it indicates that the current assignment is invalid, and the algorithm backtracks.

Advantages of the Algorithm:

- **Efficiency**: When combined with heuristics like MRV, LCV, and forward checking, the backtracking algorithm can efficiently prune the search space and avoid unnecessary exploration of invalid configurations.
- **Flexibility**: The backtracking approach can be adapted to a wide range of CSPs, not just Sudoku, making it a versatile algorithm for solving combinatorial problems.
- **Optimality**: Backtracking guarantees finding a solution if one exists, as it explores all possibilities systematically.

Disadvantages of the Algorithm:

- **Exponential Time Complexity**: In the worst-case scenario, backtracking can have an exponential time complexity as it explores all possible assignments. Although heuristics help reduce the search space, the algorithm may still be slow for very large or complex CSPs.
- **No Guarantees on Performance**: While the algorithm will find a solution (if one exists), it is not always the most efficient in terms of time or memory usage, especially for puzzles with high constraint density or large search spaces.

In this task, the backtracking algorithm, enhanced with forward checking and heuristics, provides an efficient way to solve Sudoku puzzles by minimizing the number of steps required to reach a valid solution.

Implementation

The implementation of the code is the following.

1. We first make a class called *CSP* and initialize it:

```
class CSP:
    def __init__(self, variables, domains, constraints):
        """
        Initialize the CSP for solving Sudoku.
        """         self.variables = variables
self.domains = domains         self.constraints =
constraints         self.solution = None
self.viz = []  # To store visualization steps
```

So as we mentioned in the introduction we have as initial parameters the *variables*, which are the cells/positions of the Sudoku grid (e.g. the first position of the grid is (0, 0)), the *domains* which are the available values for each cell (e.g. for the case of an empty cell the domain is [1, 2, ..., 9]), and also the known *constraints*. We have also defined the *solution*, and the parameter *viz* for the visualization steps.

2. Function for printing the Sudoku grid

```
    def print_sudoku(self, puzzle):
        for i in range(9):
if i % 3 == 0 and i != 0:
                print("- - - - - - - - - - - ")
for j in range(9):                 if j % 3 ==
0 and j != 0:                     print(" | ",
end="")                 print(puzzle[i][j],
end=" ")             print()         print("\n")
```

A basic function to print the Sudoku puzzle/grid.

3. The solve function

```
    def solve(self):
        """
        Solve the Sudoku puzzle.
        """         assignment = {}         self.solution =
self.backtrack(assignment)         return self.solution, self.viz
```

The solve function is created for a cleaner code, and it implements the backtrack function that we created, also returning the corresponding solution and visualization steps.

4. Functions used in the backtracking function

```
    def is_valid(self, var, value, assignment):
        """
        Check if assigning 'value' to 'var' is valid according to constraints.
        """         r, c = var         for (x, y) in assignment:             if
assignment[(x, y)] == value:                 if x == r or y == c or (x // 3 ==
r // 3 and y // 3 == c // 3):
                    return        False
return True
```

The *is_valid* function has as parameters the variable (*var*), meaning of course a certain position on the gird, the *value* which is a certain value assigned to a variable, and the *assignment* which is a dictionary with all the assignments to the variables. The function checks if a value can be assigned to a variable according to the known constraints, and returns True or False accordingly.

```
    def least_constraining_value(self, var):
        """
        Sort values based on how much they constrain future choices.
        """
        return     sorted(self.domains[var],      key=lambda   v:
self.count_conflicts(var, v))



    def count_conflicts(self, var, value):
        """
        Count how many variables would lose this value from their domain.
        """         r, c =
var         conflict_count
= 0
                for (x, y) in
self.variables:
            if (x == r or y == c or (x // 3 == r // 3 and y // 3 == c // 3))
and (x, y) != var:                 if value in self.domains[(x, y)]:
                    conflict_count += 1   # This value removes an option from
another variable                 return conflict_count   # Higher means worse,
lower means better choice
```

These helper functions are used in the backtracking function in order to decide which of the available values of a variable we can assign, are the most efficient based on the number of conflicts it would cause. So we are not just assigning a random or the smallest value of the domain in the steps of backtracking, but we assign the one that will cause the fewer changes of the domains of the other variables that are constrained by it. Specifically, in the *count_conflicts* function, we check if the values of the domain of the variable exist in the other variables as well. And we count it as a conflict if the other variables also have that available value in their domain, meaning that if we use the value, they will lose that available value from their domain in the forward checking process causing consequently a conflict. Finally, in the *least_constraining_value* function we return the value of the domain that cause the fewer conflicts.

5. Forward checking

```
    def forward_checking(self, var, value, assignment):
        r, c = var
        pruned_values = []  # Store removed values for restoration

        for (x, y) in self.variables:
            if (x == r or y == c or (x // 3 == r // 3 and y // 3 == c // 3))
and (x, y) not in assignment:
                if value in self.domains[(x, y)]:
                    self.domains[(x, y)].remove(value)
pruned_values.append((x, y, value))
                    if not self.domains[(x, y)]:   # If a domain is empty,
backtrack                            return False, pruned_values  # Return pruned
values

        return True, pruned_values
```

With the *forward_checking* function having as parameters again the variable, the value and the assignment, we remove the value that we used from the domains of the variables of the grid that are constrained by the variable, its value we changed. This way we prune the values that will not be allowed later in the algorithms because of the constraints. As a result we manage to reduce the number of invalid paths explored, detect conflicts early, and prune the search space. This allows the algorithm to reach a solution faster and makes it more scalable to larger and more complex Sudoku puzzle.

6. Backtracking

```
    def backtrack(self, assignment):
        if len(assignment) == len(self.variables):   # If all variables are
assigned            return assignment

        var = min(self.variables, key=lambda v: len(self.domains[v]) if v not
in assignment else float('inf'))
        for value in self.least_constraining_value(var): # We try the least
constraining value first            if self.is_valid(var, value, assignment):
                assignment[var] = value
                self.viz.append((var, value))  # Store for visualization

                success, pruned_values = self.forward_checking(var, value,
assignment)                if success:
                    result = self.backtrack(assignment)
if result:
                        return result

                # Now if we have a deadend, we need to
backtrack                # Restore pruned values when
backtracking                for x, y, v in pruned_values:
self.domains[(x, y)].add(v)

                self.domains[var] = {v for v in range(1, 10) if
self.is_valid(var, v, assignment)}

                del assignment[var]  # Undo assignment

        return None
```

In the *backtrack* function we basically implement how the algorithm solves the puzzle using backtracking and forward checking. Basically, we iteratively call this function until the puzzle is solved. We start by checking if the length of the assignment is the same as the length of the variables. If that's the case the algorithm has reached a solution, which we then return as the final assignment. If not, we find the variable that has the least available values to assign in its domain. This is the Minimum Remaining Values (MRV) heuristic that selects the variable with the fewest remaining legal values in its domain, which helps to reduce the search space and guides the algorithm to make more promising decisions early on. Then, from that variable, using the Least Constraining Value (LCV) heuristic we choose to select the value that as we mentioned leads to the minimum conflicts. After that, if the value is valid, we assign it to the variable, we append the pair (variable, value) to the visualization, and we execute forward checking. If we are successful we continue executing the backtracking function, otherwise if we reach a deadend we backtrack to a previous step so that we can assign another value to the

variable. If that's the case we also update the domains of the affected variables by adding back the pruned values and we delete the unsuccessful assignment. This is basically how the whole algorithm works!

7. Visualization

```
    def visualize(self):
        """
        Visualize each step's board state with step numbers
        """         if
not self.viz:
            print("No steps to visualize.")
return

        board = [[puzzle[i][j] for j in range(9)] for i in range(9)]  # Copy
original puzzle                 for step, (var, value) in enumerate(self.viz,
1):
            row, col = var
            board[row][col] = value  # Update visualization board

            print('#' * 8 + f' Step {step} ' + '#' * 8)
self.print_sudoku(board)
```

This function is used for the visualization of each step of the process of the algorithm, and it prints the Sudoku board in every step.

8. Initialization of the Sudoku problem

```
variables = [(r, c) for r in range(9) for c in range(9) if puzzle[r][c] == 0]
domains = {} for r in range(9):     for c in range(9):         if puzzle[r][c]
== 0:
            # Compute allowed values based on row, column, and box constraints
used_values = set(
                puzzle[r] +  # Row values
                [puzzle[i][c] for i in range(9)] +  # Column values
```

```
                [puzzle[i][j] for i in range(r//3 * 3, (r//3 + 1) * 3)
for j in range(c//3 * 3, (c//3 + 1) * 3)]  # Box values            )
            domains[(r, c)] = set(range(1, 10)) - used_values

constraints = []
for r in range(9):
    constraints.append([(r, c) for c in range(9)])
for c in range(9):
    constraints.append([(r, c) for r in range(9)])
for box_r in range(0, 9, 3):     for box_c in
range(0, 9, 3):
        constraints.append([(box_r + r, box_c + c) for r in range(3) for c in
range(3)])
```

This code initializes the Sudoku Constraint Satisfaction Problem (CSP) by defining variables, domains, and constraints. The variables represent the empty cells in the Sudoku grid, identified as coordinate pairs. The domains store the possible values (1-9) for each empty cell, determined by eliminating numbers already present in the same row, column, and 3×3 box. The constraints ensure that each row, column, and box contains unique values by grouping related cells together. This setup allows the CSP solver to systematically assign values while maintaining Sudoku's fundamental rules, forming the foundation for an efficient backtracking search with forward checking.

Finally, we also need to add a function that deals with conflicts such as having the same numbers in either columns, rows or 3x3 grids.

```
def validate_puzzle(puzzle):
```

Takes one argument, puzzle, which is expected to be a 9×9 list of lists.

Its goal is to check that all the pre-filled cells (non-zero numbers) follow Sudoku rules—specifically, that no row, column, or 3×3 box has duplicate values.

```
    # Check rows
    for i, row in enumerate(puzzle):
        nonzeros = [num for num in row if num != 0]
        if len(nonzeros) != len(set(nonzeros)):
            print(f"Conflict found in row {i+1}: {row}")
            return False
```

The code loops over each row using *enumerate*, so i is the row index and row is the list of numbers in that row.

The list comprehension [num for num in row if num != 0] creates a list called nonzeros containing only the non-zero numbers (i.e., the pre-filled cells).

It compares the length of nonzeros with the length of set(nonzeros).

- A **set** automatically removes duplicates.

- If there are duplicates, the set will be smaller than the list.

If a conflict is detected (i.e., the lengths differ), it prints a message indicating which row has the conflict and returns False.

```
    # Check columns
    for c in range(9):
        col = [puzzle[r][c] for r in range(9) if puzzle[r][c] != 0]
        if len(col) != len(set(col)):
            print(f"Conflict found in column {c+1}: {col}")
            return False
```

The loop iterates over each column index c from 0 to 8.

A list comprehension gathers the value at position [r][c] for each row r (only if the value is not 0). This list, col, represents the non-empty values in that column.

Same logic as rows applies.

```
    # Check 3x3 boxes
    for box_r in range(0, 9, 3):
        for box_c in range(0, 9, 3):
            box = []
            for r in range(box_r, box_r+3):
                for c in range(box_c, box_c+3):
                    if puzzle[r][c] != 0:
                        box.append(puzzle[r][c])
            if len(box) != len(set(box)):
                print(f"Conflict found in 3x3 box starting at
({box_r+1},{box_c+1}): {box}")
                return False
```

The outer loops iterate over the starting indices of each 3×3 box.

- range(0, 9, 3) produces the starting row indices (0, 3, 6) and similarly for columns.

For each 3×3 block, nested loops iterate through the cells within that box (from box_r to box_r+2 and from box_c to box_c+2).

The function checks whether the length of box matches the length of set(box).

If a conflict is found, it prints a message indicating the starting position of the 3×3 box and returns False.

If no conflicts are found the function returns True.

Discussion

In the previous sections, we introduced the Sudoku puzzle as a Constraint Satisfaction Problem (CSP) and presented a backtracking solution enhanced by Forward Checking and two

heuristics: the Minimum Remaining Values (MRV) and Least Constraining Value (LCV). In this section, we illustrate and discuss the performance of our solver by walking through step-by-step visualizations and exploring a variety of test cases. We also provide insights into how forward checking tangibly improves the algorithm's efficiency.

Below is the initial puzzle we focused on solving:

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

When we run our CSP solver on this puzzle, we store every assignment (var, value) in a list called viz. For each assignment, we can print the board. Here is an abbreviated sample of what the step-by-step visualization looks like:

Step 1: The algorithm finds a suitable empty cell, say (0, 2), and assigns the digit 4.

Step 2: Another cell, for instance (1, 0), gets assigned the digit 6, and so on.

We continue in this manner until every empty cell is assigned. If at any point a conflict arises (e.g., a chosen digit makes it impossible for a neighboring cell to have any legal value), forward checking detects it, and the solver backtracks to the previous assignment.

After dozens of these incremental assignments, the solver arrives at a complete solution:

The number of steps depends on puzzle difficulty and the order in which variables are chosen, but typically our heuristics significantly reduce the branching factor.

Additional Test Cases

To demonstrate the robustness of our solver, we applied it to several different Sudoku configurations:

- **A Puzzle with No Valid Solution**

Consider a puzzle with an obvious conflict in the first row (two identical digits forced by the row constraints):

5	3	0	0	7	0	0	5	0
0	0	0	1	0	5	0	0	0
0	9	8	0	0	0	0	6	0
.	.	.	.	.	.	.	.	.

←Notice how two 5s are inserted into the same row

When we run our solver, it tries assignments and systematically backtracks each time it encounters contradictory constraints. Ultimately, every path is exhausted, and no complete assignment is found. The solver thus prints:

```
Conflict found in row 1: [5, 3, 0, 0, 7, 0, 0, 5, 0]
The initial puzzle has conflicting clues! Please fix them before solving.
PS C:\Users\billd>
```

This confirms that the CSP approach gracefully handles unsolvable puzzles by checking every possibility until it can conclude no solution is possible.

- **A Nearly Solved Puzzle**

8	2	7	1	5	4	3	9	6
9	6	5	3	2	7	1	4	8
3	4	1	6	8	9	7	5	2
5	9	3	4	6	8	2	7	1
4	7	2	5	1	3	6	8	9
6	1	8	9	7	2	4	3	5
7	8	6	2	3	5	9	1	4
1	5	4	7	9	6	8	2	0
2	3	9	8	4	1	5	0	7

Because only a couple of cells are empty, the solver finishes almost instantly, with only a handful of steps required. This shows that forward checking and heuristics are especially beneficial for quickly finalizing partially completed puzzles.

- **Puzzle With Multiple Solutions**

This test case uses a Sudoku grid with far fewer than the typical 17 clues. As a result, the puzzle is under-constrained and can lead to several valid complete solutions.

5	0	0	0	0	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	0
0	0	0	0	0	0	0	0	6
0	0	0	0	0	0	2	0	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

With only 12 clues, the grid does not restrict the number of valid solutions. In standard Sudoku, 17 clues are typically needed for a unique solution. When run on this puzzle, the solver will find a complete assignment that satisfies all Sudoku constraints. However, since there are multiple solutions, the solver will report the first one it finds. We could have also searched for all the possible solutions if we had included it in our code.

Conclusion

In this lab, we successfully modeled Sudoku as a Constraint Satisfaction Problem (CSP) and implemented a backtracking solver with forward checking and **two heuristics** (MRV and LCV). Specifically, we showed how:

1. **Forward Checking** prunes the domains of neighboring cells as soon as we make an assignment, reducing the risk of pursuing dead ends.
2. **MRV** helps us choose the cell with the fewest possible values first, preventing deep exploration into obviously harder branches.
3. **LCV** picks each digit in an order that leaves the maximum flexibility for future cells, further improving efficiency.

Through several test cases we saw how the algorithm adapts to different scenarios. We confirmed the solver can either produce a valid solution or conclusively determine that no solution exists.

Key lessons and potential enhancements include:

- **Heuristic Ordering**: Combining MRV, LCV, and forward checking already yields strong performance, but advanced techniques could further prune domains.
- **Complexity**: Despite heuristics, backtracking can still become expensive for extremely sparse or large puzzles. However, for standard Sudoku, the solver typically runs in negligible time.
- **User Interface**: While we currently rely on text-based visualization, future work might include a graphical interface to illustrate the solving process more intuitively.

Overall, this lab demonstrated how **classical CSP methods** and **systematic backtracking** can elegantly tackle Sudoku. The approach is general enough to apply to other combinatorial puzzles (like crosswords, n-queens, or map coloring) that share the same notion of variables, domains, and constraints.