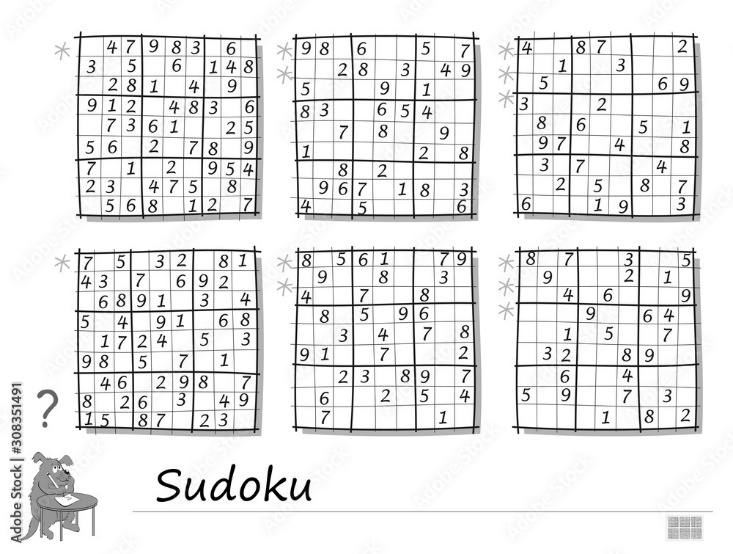
**IT159: Artificial Intelligence**

**Lab#4 & 5: Sudoku (Constraint Satisfaction Problems)**

# Introduction



In Constraint Satisfaction Problems (CSPs), the goal is to find a complete, consistent assignment of values to a set of variables 𝑋 (taken from their domains 𝐷) satisfying a set of constraints 𝐶 that limit the valid combinations of variable values. In this assignment, you will have an opportunity to develop a program using CSP solution techniques to solve Sudoku puzzles.

Sudoku (Japanese meaning *number place*) puzzle is a 9x9 grid (81 variables) where each cell in the grid can take on the integer values 1-9 (the domain of each variable). A solution to a Sudoku puzzle is an assignment of values for each cell in the grid such that no two cells in the same row, column, or 3x3 square have the same value.

For example, for an initial configuration of a Sudoku puzzle, you might be given:

**003|020|600**

**900|305|001**

**001|806|400**

**-----------**

**008|102|900**

**700|000|008**

**006|708|200**

**-----------**

**002|609|500**

**800|203|009**

**005|010|300**

which has the solution:

**483|921|657**

**967|345|821**

**251|876|493**

**-----------**

**548|132|976**

**729|564|138**

**136|798|245**

**-----------**

**372|689|514**

**814|253|769**

**695|417|382**

# Getting Started

The code you will be using can be downloaded as a zip archive on Blackboard, namely **Lab45-Sudoku**.

# Assignment

Your assignment is to write a program in Python that can take a set of Sudoku puzzles as input from a file, models each puzzle as a CSP, and outputs solutions to each puzzle.

Extract the files into a directory/folder on your computer which will contain several files of Sudoku puzzles:

1. data/euler.txt, a set of Sudoku puzzles from Project Euler https://projecteuler.net/problem=96

2. data/magictour.txt, a more difficult set of Sudoku puzzles from http://magictour.free.fr/top95 Each file contains a multiple Sudoku puzzles (one per line), in the following format:

* Each line is a string of 81 characters, where characters in positions 0-8 correspond to the first row of the puzzle, characters in positions 9-17 correspond to the second row of the puzzle, etc.
* Known values are represented by the digits 1-9.
* Initially unknown values are represented by digit 0.

And other files that you can and cannot modify as below

**Files you'll edit:**

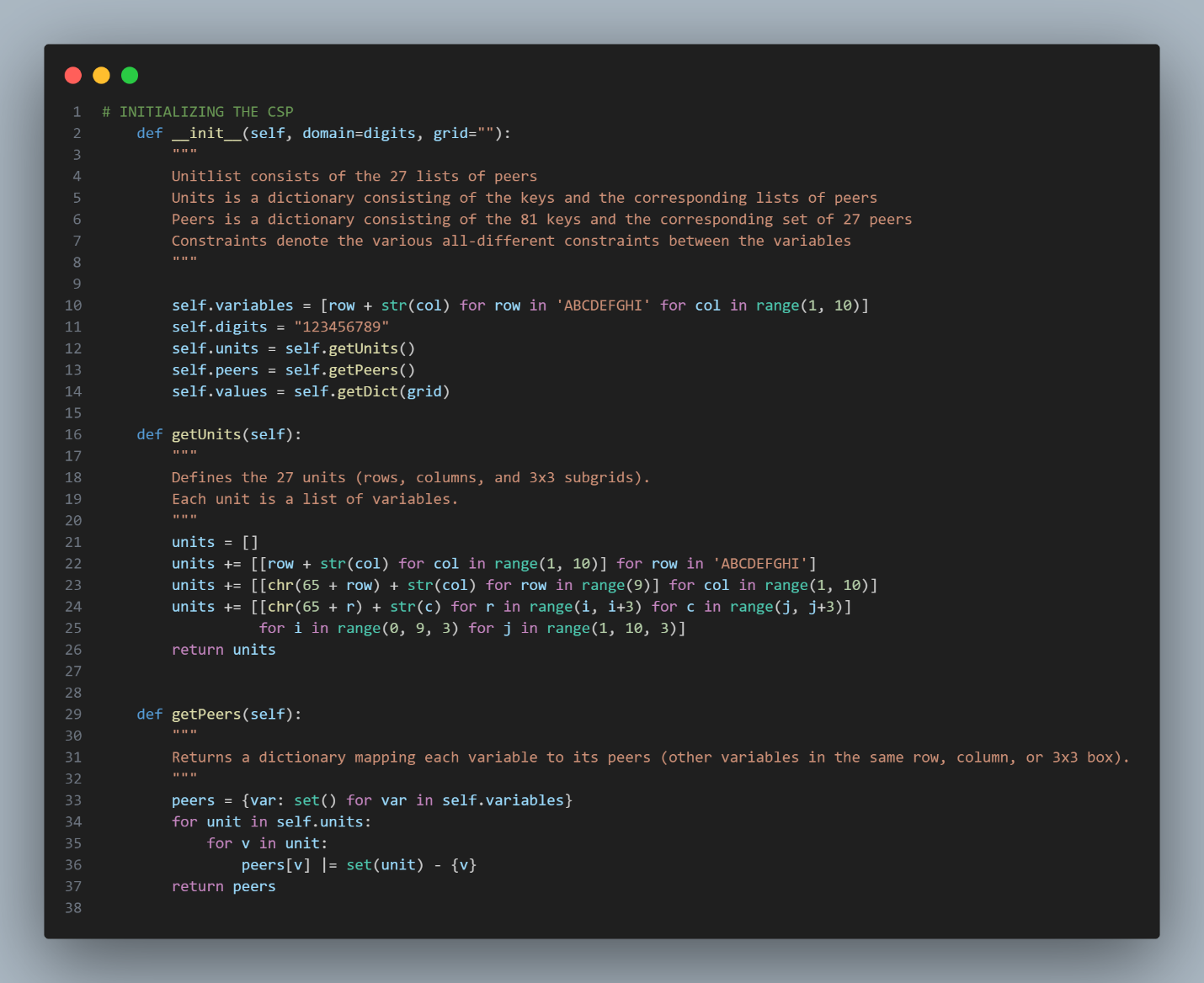
search.py Where all your search algorithms will reside.

csp.py Class description for constraint satisfaction problem.

**Files you should look at but NOT edit:**

|  |  |
| --- | --- |
| util.py  sudoku.py | Useful data structures for implementing search algorithms.  The main file that runs to solve all Sudoku problems. |

**Exercise 1**: Implement the constraint satisfaction problem in the initialize function in the csp.py. How did you represent the Sudoku puzzle a CSP? What design options did you consider, and how did you decide on this implementation?



A **Constraint Satisfaction Problem (CSP)** consists of:

* **Variables** → 81 grid cells labeled using row-column notation (e.g., A1, B3).
* **Domains** → Each variable (cell) has a domain {1-9}, except for pre-filled cells which have a fixed value.
* **Constraints** → Each row, column, and 3×3 subgrid must contain **all different numbers** (no duplicates).

In **csp.py**, I initialized:

* variables: The set of 81 grid positions {A1, A2, ..., I9}.
* domains: A dictionary mapping each variable to {1-9} or a fixed value if pre-filled.
* constraints: Ensured that **each variable shares constraints with its row, column, and 3×3 box peers**.

**Option 1: Direct 2D Array Representation**

* Store the Sudoku grid as a **9×9 array** (grid[row][col]).
* Use row/col indices to enforce constraints.  
  **Pros**: Simple to implement.  
  **Cons**: Harder to track dependencies across constraints.

**Option 2: Dictionary-Based CSP Representation (Chosen Approach)**

* Represent variables as {A1, A2, ..., I9}.
* Store domains as {A1: "123456789", B3: "5", ...}.
* Maintain **peer relationships** (row, column, and 3×3 box constraints).  
  **Pros**: More flexible, easy constraint enforcement.  
  **Pros**: Scalable for constraint propagation (AC-3, Forward Checking).  
  **Cons**: Slightly more complex data structure.

**Exercise 2**: Implement Backtracking Search algorithm in the search.py.



**Exercise 3**: Implement AC-3 search algorithm.



Your program should be able to read in these puzzles, solve them, then output the solutions in the same format (a string of 81 digits, followed by a newline character) in the same order they were read in from file, so that it is called as follow:

python3 sudoku.py -–inputFile <PuzzleFile>





1. In file sudoku\_solution.txt

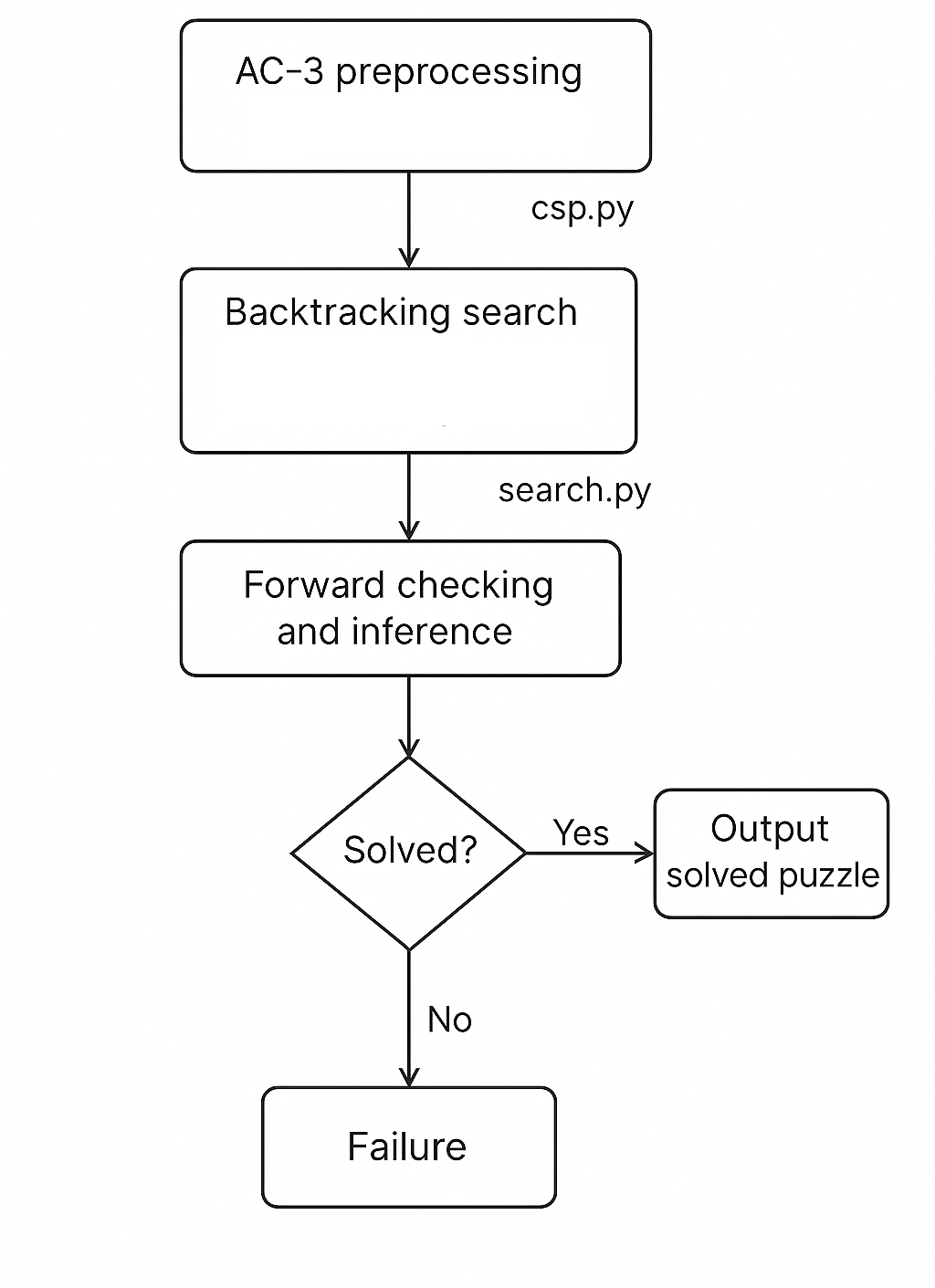
2.

During the assignment, I have struggled with optimizing the algorithm to efficiently handle large problems, as well as converting the Sudoku problem into CSP form for solving. In particular, I had some difficulty working with data files and exporting the results in the required format. However, after some research and experimentation, I was able to complete the course and feel proud of the results.

3.

It took me 2-3 days to complete the assignment.

4. Pipline Analysis



First, **AC-3** preprocessing ensures arc consistency by reducing variable domains, removing inconsistent values early. Next, the **Backtracking Search** algorithm assigns values to variables using heuristics like Minimum Remaining Values (MRV) and the Least Constraining Value to prune the search space, and backtracks when a conflict occurs. **Inference** is applied via forward checking, dynamically reducing domains of neighboring variables as values are assigned, further pruning possibilities. The search terminates when all variables are assigned consistent values that satisfy the Sudoku constraints or when no solution is found. Finally, the pipeline is analyzed for correctness, performance, and scalability, with improvements like heuristic tuning or parallelization considered to optimize the solution.

# What to submit

1. The solutions to all 145 puzzles in the same format as the input files (please put all of the euler.txt solutions under a header called “Euler” and the magictour.txt solutions under a header called “Magic Tour”)
2. A short paragraph describing your experience during the assignment (what did you enjoy, what was difficult, etc.)
3. An estimation of how much time you spent on the assignment.
4. Source code + README (how to compile and run your code).
5. Please create a folder called "yourname**\_**StudentID\_Lab45" that includes all the required files and generate a zip file called "yourname\_StudentID\_Lab45.zip".
6. Please submit your work (.zip) to Blackboard.