Sudoku Solver as Constraint Satisfaction Problem vs Backtracking

Submitted by Amit Ben Zaken

Project summary

This project implements a Sudoku solver by casting the classic 9×9 puzzle as a Constraint Satisfaction Problem (CSP) and compares its performance to the regular backtracking solve approach. As learned in the lecture, a CSP is defined by:

- Variables (X): the unknowns to solve for.
- Domains (D): the set of possible values each variable may take (in this case I used individual domains for variables).
- Constraints (C): relations that restrict the simultaneous assignments of values to variables.

By representing each Sudoku cell as a CSP variable, its possible digits as the domain, and the puzzle rules (rows, columns, 3×3 blocks contain all-different digits) as constraints, we can use a generic CSP engine to find a valid filling that completes the puzzle.

Implementation of CSP

1. Variables: X = [(row, col) for row in range(9) for col in range(9)]
   Each variable is represented by a tuple of its location on the board.
2. Domain:
   Each variable got a domain according to if it is in the board were solving. If that slot is vacant the domain is {1-9} but if it’s taken, then the variable can have that value only so its domain would be the singleton.
3. Constraints:
   We can use “all-different” constraints for the rows, columns and 3x3 blocks.
4. Solver:
   We use the python-constraint library’s Problem() object to define variables, domains, and constraints, then call getSolution() to get the result.

Example run

	Input	CSP	Backtracking
Solution	____7____ 6__195___ _98____6_ 8___6___3 4__8_3__1 7___2___6 _6____28_ ___419__5 ____8__79	534678912 672195348 198342567 859761423 426853791 713924856 961537284 287419635 345286179	345678912 672195348 198342567 859761423 426853791 713924856 961537284 287419635 534286179
Average Time (seconds, 100 iterations)	N/A	0.0025	0.1080

Conclusion – We can see CSP performs significantly faster than backtracking.