Task 1

Formulate the Sudoku puzzle as a constraint satisfaction problem (CSP).

A detailed description of all variables, domains and constrains that are sufficient to model Sudoku as a CSP.

A Sudoku board consists of a grid of 81 *cells,* along 9 rows and 9 columns, partitioned into 3-cell by 3-cell *sections,* with each cell containing a number in the closed range [1-9]. Each cell can be represented with a variable, C_{11} through C_{99}, using a matrix-like two-index notation. For the sake of brevity, these 3x3 sections will be referred to as S_{11} through S_{33}, using the same matrix-like notation.

C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}
C_{21}	C_{22}	C_{23}	C_{24}	C_{25}	C_{26}	C_{27}	C_{28}	C_{29}
C_{31}	C_{32}	C_{33}	C_{34}	C_{35}	C_{36}	C_{37}	C_{38}	C_{39}
C_{41}	C_{42}	C_{43}	C_{44}	C_{45}	C_{46}	C_{47}	C_{48}	C_{49}
C_{51}	C_{52}	C_{53}	C_{54}	C_{55}	C_{56}	C_{57}	C_{58}	C_{59}
C_{61}	C_{62}	C_{63}	C_{64}	C_{65}	C_{66}	C_{67}	C_{68}	C_{69}
C_{71}	C_{72}	C_{73}	C_{74}	C_{75}	C_{76}	C_{77}	C_{78}	C_{79}
C_{81}	C_{82}	C_{83}	C_{84}	C_{85}	C_{86}	C_{87}	C_{88}	C_{89}
C_{91}	C_{92}	C_{93}	C_{94}	C_{95}	C_{96}	C_{97}	C_{98}	C_{99}

The domains of each of these variables is identical. $D(C_n) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Informally, no number may appear multiple times in the same row, column, or section. Formally, this defines 8 constraints per row, 8 constraints per column, and 8 constraints per section. All 216 constraints could be enumerated explicitly, however, that would make this document far too long. These constrains are defined generally

In the following definitions, let:

$$\mathbb{D} = 1, 2, 3, 4, 5, 6, 7, 8, 9$$

$$a, b, c, d, i, j, k \in \mathbb{D}$$

such that:

$$C_{ij} \neq C_{kj}$$

$$C_{ij} \neq C_{ik}$$

$$\text{if } C_{ab} \in S_{ij} \wedge C_{cd} \in S_{ij}, \text{ then } C_{ab} \neq C_{cd}$$

Task 4

Specific performance data is made available in a couple of formats as `data.csv`, `data.json`, and `data.pdf` (generated from `data.md`). This performance data was collected using hyperfine (link) version 1.18.0, a rather nice CLI benchmarking tool, using a sample size of $20,000$ runs for each command. The JSON data may be visualized using some scripts available on the hyperfine repository, however, I did not see much value in that. The data set is included for completeness. As for system information as a reference, the system on which these benchmarks were performed was an Intel i7-10870H running 64-bit GNU Linux (specifically an Ubuntu 22.04 derivative).

The program is fairly deterministic, directly using no random number generation. Thus the primary source of variation between runs of the same command would be the presence of other processes on the system.

As backed up by the data, the smart backtracking algorithm outperformed the simple backtracking algorithm for each puzzle. Most notably, for the "evil" puzzle, the smart backtracking algorithm outperformed the simple backtracking algorithm by an order of magnitude.

To analyze the performance improvement for each puzzle, the speedup between the simple and smart backtracking algorithms will be computed as described below:

Let:

$$P = \text{mean execution time of simple algorithm}$$

$$S = \text{mean execution time of smart algorithm}$$

$$D = \%\text{ decrease} = 100 \cdot \frac{P - S}{P}$$

Puzzle	Simple Mean Time (ms)	Smart Mean Time (ms)	Speedup
Easy	3.7	3.0	18.9%
Medium	4.4	3.3	25.0%
Hard	4.5	3.5	22.2%
Evil	41.6	7.1	82.9%

The simple backtracking algorithm is a pure guess-and-check approach. The first unassigned variable is selected, and is assigned the first value which does not violate any constraints. This process is repeated until an unsatisfiable state is reached. At that point, the algorithm backtracks, undoing the last assignment, and trying the next value for that variable. If all possible values for any variable are exhausted, the algorithm backtracks again. Theoretically, this should solve any solvable Sudoku, as it will essentially perform a depth-first search of the entire set of possible assignments. This algorithm, while correct, is unoptimized.

The smart algorithm performs its search in the exact same way, but before each guess will perform all assignments to variables whose domain has been constrained to a single value. This assignment to the most constrained variables is a simple first step humans will use when solving Sudoku puzzles. If 1, 4, 5, and 8 appear on the same row, 2, 3, and 7 appear on the same column, and 6 appears in the same 3x3 section, then that cell must contain a 9.

These most-constrained variables are determined and are assigned their values repeatedly until a none such variables remain. At that point, the algorithm will perform one guess as described above, and will then check again for most-constrained variables. When backtracking, the smart algorithm will backtrack to the last guess made, skipping the forced variable assignments.

This improvement greatly reduces guessing, and significantly reduces the search space. The "trivial" puzzle which is also included in the input data is a puzzle with can be solved purely by assigning most-constrained variables with no guesses made.