1. Approach taken

The sudoku solver reduces the sudoku to an exact-cover problem and solves it with Algorithm X using a simplified version of Dancing Links [ref]. The reduction to an exact-cover problem builds all sudoku constraints (that each row, column, and sub-grid consists of all digits from 1 to 9 exactly once) and their qualifying candidates for each cell into a compact and structured tabular representation, further explained below. This transformation enables faster elimination and backtracking since every candidate value for each cell is mapped to the constraints it must satisfy using a simple HashMap data structure. Computational power is substantially saved via efficient pruning of the search space and the reduced need to remember various search paths.

2. Description of Algorithm

The high-level logic of the solver is inspired by the theoretical framework and pseudo code in Harrysson and Laestander's (2014) degree project and the implementations of Blidh (2000). The fundamental step of this algorithm is reducing a sudoku to an exact-cover problem (Kapanowski, 2011), described below. Here, the two Python dictionaries `cell_constraints` and `self.map` under class `SudokuSolver` play a crucial role. `cell_constraints` maps cell-value candidate pairs to their relevant constraints, while `self.map` maps each constraint to each candidate cell-value pair. For instance, for every coordinate (r, c) where r denotes the row and c denotes the column, it initially has 9 value candidates, denoted by (r, c, n), where $n = 1, 2, \ldots 9$. Each of these candidates are subject to four constraints represented in tuple structure as follows:

("notEmpty", (r, c)) – that the cell at coordinate (r, c) cannot be empty; ("uniqRow", (r, n)) – that the value n can only appear once in row r; ("uniqCol", (c, n)) – that the value n can only appear once in column c; ("uniqBox", (b, n)) – that the value n can only appear once in box b, where b denotes the index of the 3×3 sub-grid the cell at (r, c) belongs to.

It precisely these four constraints that makes the sudoku qualify as an exact-cover problem. Each of these constraints in turn have a list of nine initial candidates. For example, for any non-empty constraint ("notEmpty", (r, c)), it initially should have 9 candidates, expressed as (r, c, n), where $n = 1, 2, \ldots 9$; any unique-row constraint ("uniqRow", (r, n)) should also have 9 initial candidates (r, c, n) where $c = 1, 2, \ldots 9$; any unique-column constraint ("uniqCol", (c, n)) should have 9 initial candidates (r, c, n) where $r = 1, 2, \ldots 9$; any unique-subgrid constraint ("uniqBox", (b, n)) should have 9 initial candidates (r, c, n) where each (r, c) correspond to the 9 cells in box index b. The recursive solving process can be summarised as:

Step 1: Select a candidate $m = (r, c, n)$ and using it as the key to find all its four constraints ($\boldsymbol{C_i}$ *for* $i = 1, 2, 3, 4$) in `cell_constraints`.

Step 2: For each $\boldsymbol{C_i}$, eliminate all candidates in `self.map` that contradicts $\boldsymbol{C_i}$, given that we assume m to now be a populated cell. Let us denote these contradicting candidates as set $\boldsymbol{T}$. We know $\boldsymbol{T}$ is already contained in all the candidates currently stored under key $\boldsymbol{C_i}$ in `self.map`. However, it is important to understand this is only one appearance amongst of potentially many, such that they could also be present under <u>other</u> constraint keys in `self.map`. Hence, the `select()` function performs a full sweep across `self.map` and the eliminates all <u>other</u> presences of candidates in $\boldsymbol{T}$, other than those currently stored under key $\boldsymbol{C_i}$.

Step 3: Remove $\boldsymbol{C_i}$ and all its associated candidates from `self.map` and save them in a list called `removed_candidates`. What is saved here is a combination of $\boldsymbol{T}$ plus the candidate m selected in Step 1.

Finally, backtracking is made extremely easy as all there is to do is to find the last four sets appended to `removed_candidates`, go through every set to retrieve each removed candidate, and revert the elimination process in Step 2 above, restoring all deletions previously made, as implemented in the `revert()` function.

3. Optimisations and Complexity

The two dictionaries mentioned above underpins the optimisations of this algorithm, building a two-way linkage between each candidate value for every individual cell with all granular constraints of the sudoku. The values of `self.map` also adopts the set data structure. Therefore, elimination become no other than making searches through HashMaps and HashSets, both of which typically has a O(1) time complexity. Moreover, throughout the recursive solving function, the key-value pairs in `cell_constraints` remain a constant lookup-table, which plays an important role in preserving meta-data for each candidate such that backtracking can be completed without remembering particular sequence of events.

4. Reflections and suggestions

Most puzzles took less than 10 milliseconds to solve, except for hard sudoku number 5 and 10 taking 30 and 10 milliseconds. The results are very much satisfactory for this particular usage. However, the extendibility of the algorithm to other search problems remains questionable, as not of them can be reduced to an exact cover problem. Moreover, the necessity to store all constraints and candidates can require large memory consumption, suggesting it may not scale well to larger sudokus.

References:

Andrew Dunstan 2020. *Sudoku Solver* [Online]. Available from: https://www.youtube.com/watch?v=UqGj9iAcB4M [Accessed 16 March 2024].

Blidh, H. 2020. Dlxsudoku. GitHub [Online]. Available from: https://github.com/hbldh/dlxsudoku/tree/master/dlxsudoku [Accessed 15 March 2024].

Kapanowski, A., 2011. Python for education: the exact cover problem. *The Python Papers, 6*(4). Available from https://doi.org/10.48550/arXiv.1010.5890

Knuth, D. 2000. Dancing Links. *Millennial Perspectives in Computer Science*, pp. 187-214.

Harrysson, M., & Laestander, H., 2014. *Solving Sudoku efficiently with Dancing Links.* Degree Project in Computer Science, DD143X. Stockholm University.

SIGma 2023. *[SP23] Meeting 4 - Algorithm X* [Online]. Available from: https://www.youtube.com/watch?v=iv6pB4WCFIo [Accessed 15 March 2024].

Stanford Online 2018. *Stanford Lecture: Don Knuth—"Dancing Links" (2018)* [Online]. Available from: https://www.youtube.com/watch?v=_cR9zDlvP88&t=1856s [Accessed 15 March 2024].