CS3243 Assignment 2 **Group A2_34**
Name: Cheong Siu Hong Matric. No.: A0188018Y
Name: Douglas Wei Jing Allwood Matric. No.: A0183939L

Setup

In Sudoku, a 9x9 Puzzle grid is provided where a single player must fill up each square with a value from 1 to 9 such that no row, column, or block contains repeated digits.

Terminology used:

- **Puzzle**: A list of *lists of integers* where each *list of integers* represents a *row* of the Sudoku grid
- **Value**: An integer from 1 to 9, inclusive of both.
- **Square**: A single grid unit in the Sudoku grid that is to be filled with a Value
- **Domain**: The remaining valid Values that a Square can take
- **Block**: A non-overlapping 3x3 grid of squares. There are exactly 3 Blocks per row
- **Unit**: A collection of nine squares that are related by the *AllDiff* constraint. A unit can represent a row, column or block. Each Square belongs to exactly 3 Units.
- **Peers**: A square's peers are all the **other** squares that are in the same Units as the Square. In other words, the peers of a square are all other squares that are related to it by the *AllDiff* constraint.
- **Assigned**: A square is considered **assigned** if and only if it has a domain of size 1.
- **Invalid:** A domain is considered **invalid** if it has a size of 0.

Constraints:

- *AllDiff*: Every square in a unit must have different values.

Data Structures:

- Used a Python Dictionary (Hash-Map of Key => Value pair) to represent and manipulate the Puzzle.
  Key: 2-tuples of (row_index, column_index), each representing a square.
  Value: The remaining values that the square can take, stored as a string of integers.

Explanation

Our algorithm is split into two phases, pre-processing and search.

Pre-processing phase: The algorithm uses constraint propagation to reduce the domains of the unassigned squares to make them domain-consistent with the assigned squares in the input.

Searching phase: The algorithm uses backtracking/DFS, along with a few heuristics and optimizations.

1. Min-value heuristic. Select the square with the smallest domain size that is not yet assigned and assign a value to it. Each possible value of the square represents a possible branch in the DFS and thus this minimizes the amount of branching required.
2. Constraint propagation: Like the pre-processing step, domain consistency is ensured after any given square is **assigned**. If any assigned value causes the square domains to be **invalid**, the assignment is impossible and DFS terminates early. This ensures that the DFS will not unnecessarily recurse into branches that are definitely incorrect.
3. Unit value assignment: If any unit (row, column or block) has only one available square for a value, then assign the value to that square. This check and assignment are executed after constraint propagation at every step and may chain into further domain reductions. This optimization allows us to further narrow the search space.

Constraint Propagation:

1. Enqueue all Squares that have been assigned a value (i.e. Domain of size 1)
2. For each of these Squares, update the Domains of all their Peers (Squares that share a Unit with the Square)
3. If a Peer is updated and has a new Domain of size 1, it is now considered **assigned** and is enqueued.
4. Repeat Steps 1 to 3 until the queue is empty.
5. Call Backtrack Search with the reduced domains. Assuming the input puzzle is always solvable, the reduced domains from the initial propagation will not be **invalid.**

Backtrack Search:

1. Test if the current square domains is **invalid**. If it is then return a false value and backtrack.
2. Goal test the current square domains. If all squares are assigned then return the square domains, we are done.
3. Select a square from the square domains using the min-value heuristic.
4. For each value in the domain of this square:
    1. Assign the value to the square by removing all other values from its domain.
    2. Reduce the square domains using constraint propagation as above.
    3. Further reduce the square domains using unit value assignment.
    4. Call backtrack search with the new reduced domains.

Backtracking is necessary as it is sometimes impossible to proceed without guessing a value. When Backtrack Search guesses a value and this leads to an invalid Puzzle state, it backtracks and tries a different value. Essentially, our algorithm only guesses when it cannot make any deterministic moves, as guessing is computationally inefficient for a game like Sudoku.

Analysis

The initial preprocessing propagation is executed only once and is bound by the number of squares. As the number of squares is constant, we can say that this step runs in constant time and is negligible compared to the runtime of the searching algorithm.

For the searching algorithm, it is difficult to find a good bound for the runtime given the complexities introduced by the recursive heuristics and optimizations, but in the hypothetical worst case:

Let **n** be the number of rows = number of columns = number of values

Backtrack Search	**Constraint Propagation**
Branching factor $b = 9 \rightarrow O(n)$ Depth of search $d = 81 \rightarrow O(n^2)$	We can propagate constraints at most once from each square and there are 81 squares $\rightarrow O(n^2)$ Each assigned square has 20 neighbours $\rightarrow O(n)$ Check each unit of each neighbour: which has $3 * 9 * 27 = 27$ squares $\rightarrow O(n)$
Runtime: $O(b^d) = O(n^{n^2})$ Max number of nodes = $b^d = 9^{81}$	Runtime: $O(n^2) * O(n) * O(n) = O(n^4)$ Max steps = $81 * 20 * 27 = 43470$

Given that we run Constraint Propagation at every node, the total worst-case time complexity in Big-O notation is $O(n^{n^2} * n^4) = O(n^{n^2+4})$

Max total number of steps = $9^{81} * 43470 = 8.55 * 10^{81}$ steps

Note that the worst case time complexity stated above is an extremely loose upper bound as the hypothetical worst case scenario **cannot occur** for the following reasons:

1. The min-value heuristic minimises the number of available values at each node. The branching factor is likely to be much smaller.
2. With constraint propagation, it is possible to assign values to more than one square at each node. The depth of the search tree is likely to be smaller as well.
3. If constraint propagation loops multiple times per node, it takes a longer time but actually saves time overall in the search. This is because each loop further reduces the square domains and narrows the search space.

Therefore integrating constraint propagation into backtracking actually greatly reduces the branching factor and search depth of the backtracking search, and ends up saving a substantial amount of time in practice.