Assignment 1 - Proof of Correctness

Sriram V - CS11B058

Given an input 9x9 grid, we are to figure out if this grid can be converted into a correct sudoku solution, and if so, enumerate the steps that would perform this conversion.
The three moves allowed are:

1. Cyclically rotating columns
2. Cyclically rotating rows
3. Arbitrary permutations of 3x3 grids

Using these three moves alone, we can swap any cell with any other cell without disturbing the positions of the remaining 79 cells. – (1)

Proof:
Let us refer to the source cell as `grid[a][b][c]` where `a` is the 3x3 grid (from 0-8), and `b` and `c` are respectively the row and column within the 3x3 grid `a`. Similarly the destination cell is `grid[x][y][z]`. Then the steps to swap the two cells are as follows:

- Shift the source cell to the border of the current 3x3 grid, by swapping its position with one of the cells on the border of the 3x3 grid. The border should be the border closest to the destination grid.
- We can now perform a row(or a column) cyclic shift to push the source cell into an adjacent 3x3 grid, such that this grid is closer to the destination grid than the previous grid.
- We can ensure that the destination cell isn't affected by row or column shifts by swapping it with adjacent cells in the same grid appropriately, just before or after the row/column shift.
- Repeat the above steps until the source and destination cells are in the same grid.
- We now swap the two cells.
- Since the original state of the sudoku grid can be restored by simply performing the (already performed) steps in a reverse order, we simply need to do this to get the destination cell to the source cell's initial position (as the destination cell is now where the source cell was, before the swap occured). Thus, all other cells will be back to where they were before the moves happened.
- Hence, effectively a swap of `grid[a][b][c]` and `grid[x][y][z]` has occured.

Therefore, we can abstract out the three moves with direct cell swaps instead. Also, an inspection of a correct sudoku solution tells us that there must be only one each of 1-9 in the entire grid.

Thus, there must be 9 each of 1-9 in the entire 9x9 grid. – (2)

Since there exists atleast one correct sudoku solution, it must satisfy (2). If we are given an input grid satisfying (2), we can effectively convert this into the known correct solution because by (1) we can move any cell in the grid to any other position as required.

Hence, we can conclude that if the input grid satisfies (2) alone, we can obtain a correct sudoku solution using only the allowed 3 moves.

Thus, we check if the input grid satisfies (2). If it doesn't, then it is an invalid grid. If it does, it is a valid grid and we will be able to elucidate the steps (by expanding every required swap into a combination of the 3 moves) to convert this grid into the known, correct sudoku solution.

The Sudoku program tests for exactly this condition. If the sudoku grid is valid, the program moves values around in such a manner that exactly one each of 1-9 is present in every 3x3 grid.

A 3x3 grid may either be satisfied (already contains one each of 1-9), or may have excesses and deficiencies. For every excess that a grid has, it has a corresponding deficieny in some other value, since this excess is occupying the cell meant for another value. Thus, if we concentrate on removing only excesses from each cell, the deficiencies should also get resolved, since we will be swapping out every excess value for some other value in another 3x3 grid.

This is taken care of by the `removeExcess()` function in the program. It ensures that it removes all excesses of all values present in the current 3x3 square it is inspecting. Hence, the function also gets rid of all deficiencies. Thus, we can conclude that this function ensures that the currently inspected square is completely satisfied before moving on to the next 3x3 square. So this function makes use of an incremental design in the sense that when it is ispecting square `i`, squares `0` to `(i-1)` are completely satisfied. Thus, there is no necessity to check for suitable destination cells (for the excesses in `i`) in the `0` to `(i-1)` range. Hence all checks are performed from square `(i+1)` to `9`.

Identification of a potential destination for an excess value: We check in squares `(i+1)` to `9` whether a deficit exists in the value we are in excess of (say `val`). If it does, we next identify a potential value to swap with. This is done by checking if any of the excesses that the destination grid has is a defecit for the source grid. If we do find such a value (say `excess_val`) then we simply swap `val` (in `i`) and `excess_val` (in the destination grid). If the destination grid does have excesses but not of values the source grid is deficient in, we swap `val` (in grid i) with any `excess_val` (in the destination grid). This `excess_val` would have introduced an excess in `i` (since `i` wasn't deficient in it) and we resolve this issue by calling the `removeExcess()` function for this value.

By doing this process for every value in every 3x3 grid, we can be sure that at the end of the entire procedure only one each of 1-9 will exist in each grid.