Sudoku Solving:

**BackTracking Method**:

Input: grid size and initial layout

Output: solved problem

Step 1.

Create a list of possible values based on grid size (by default: values 1-9)

Step 2.

For each row / column / inner square create a list of possible remaining values (possible values / existing values from input)

Step 3.

Go through matrix element by element. For each element, compute the possible values (that do not violate the all different constraint) as the intersection of the possible remaining values for row, column and inner square.

If the intersection list contains only one element, that is the valid value of the Sudoku problem. Write the value at the element in the matrix and update the possible values for the row, column and inner square. Add the reevaluation of the row, column and inner square to a queue (if the queue doesn’t already contain it).

If the intersection list contains more than one element, check to see if this the number of elements in the intersection list is less that the number of the intersection from the SavePoint element. If true this element becomes the new SavePoint element

This step continues until all elements have been processed.

Step 4.

While the reevaluation queue has elements (rows, columns, inner squares), process each of them like in step 3 (for each row / column / inner square, go through all sub-elements (Sudoku squares) and execute the step 3 operation).

Step 5.

The reevaluation queue is now empty, but there are still Sudoku squares with no value.

If there is a state saved for the Sudoku board, restore it to that state and choose another value from the savepoint element. Otherwise, from the SavePoint element (it has the least number of possible values), save the state of the Sudoku board and select a random possible value.

Add the row, column and inner square of the element’s location to the reevaluation queue. And execute step 4.

Step 6. Print board

**Tabu Search**

Input: 9 x 9 sudoku board

Output: board with the least number of conflicts

Step 1:

Fill empty positions on board square by square (inner square) sequentially:

|  |  |  |
| --- | --- | --- |
| 1 | 5 | 2 |
| 7 | 3 | 4 |
| 9 | 6 | 8 |

\*red numbers represent input

Step 2: mark current board as best board.

Step 3: While max iterations not reached and there are still conflicts,

Generate neighborhood and their scores of current board (2 boards are neighbors if the only difference between could be corrected by swapping 2 numbers; score is computed by the number of conflicts / row, column and square). Sort list by score. Reset Penalty to 0; Else, step 6

|  |  |  |
| --- | --- | --- |
| 1 | 5 | 5 |
| 3 | 3 | 4 |
| 6 | 6 | 8 |

\*conflict example: 3 conflicts (3 duplicates); Input values do not suffer swapping.

Step 4: For each movement in the neighborhood, calculate penalty (frequency of movement in the long term list). If the short term list doesn’t contain the movement OR the score of the neighborhood is better than the current best board score, apply the movement to the current board, add the movement to both short and long term lists.

Step 5: If the current board’s score is better than the current best’s score, mark current one as best.

Do step 3.

Step 6: print board;

<https://github.com/Floydian888/sudoku-tabu-search>

**Backtracking + Forward Check (using all different constraint):**

Step 1: Initialize board

Step 2: Doing a depth first search in the solution space, every time a solution is assigned, we check with the remaining domain (to see if there is a violation of the all different constraint). If the domain becomes empty (there are no valid values for the next assignment), backtracking is done to the last decision point (location where assignment was made where the domain length was > 1).

Step 3: print result

<http://allantech.blogspot.ro/2007/05/solving-sudokus-as-csp-with-forward.html>