Constrain Satisfaction Problem: Sudoku Solver

1. A Sudoku puzzle can be considered a CSP.

Variables: 81 variables, one for each square / cell

Domains: Empty squares have domain {1, 2, 3, 4, 5, 6, 7, 8, 9} and the prefilled squares have a domain consisting of a single value.

Constraints: There are 27 different Alldiff constraints: one for each row, column and 3x3 box. For example:
Alldiff((0,0), (0,1), (0,2) … (0,8))
Alldiff((1,0), (1,1), (1,2) … (1,8))
…
Alldiff((0,0), (1,0), (2,0) … (8,0))
Alldiff((0,1), (1,1), (2,1) … (8,1))
…
Alldiff((0,0), (0,1), (0,2), (1,0), (1,1), (1, 2) , (2,0), (2,1), (2, 2))
Alldiff((3,0), (3,1), (3,2), (4,0), (4,1), (4, 2) , (5,0), (5,1), (5, 2))
…

2a. The Sudoku solver was implemented in Java. The source code is attached.

2b. Solutions for test puzzles

Easy:

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

Medium:

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

Hard (2 solutions):

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

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

Evil:

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

2c. Algorithms' performance

Each test was repeated 50 times, unless specified otherwise.

Table 1. Time (ms)

	B	B+FC	B+FC+H
Easy	1,044.86 +/- 1,926.36	237.52 +/- 293.63	14.4 +/- 8.72
Medium	49,057.70 +/-	10,106.54 +/-	15.7 +/- 10.77

	47,496.12 (*10 tests*)	11,979.96	
Difficult	Timeout	21,415.5 +/− 12,353.099305437481 (*10 tests*)	21.2 +/− 15.32
Evil	Timeout	87,578.8 +/− 93,443.09 (*10 tests*)	32.1 +/− 21.08

Table 2. Number of Nodes Expanded

	B	**B+FC**	**B+FC+H**
Easy	685,780.96 +/− 1,256,444.77	4,134.6 +/− 5,383.7	53.7 +/− 4.45
Medium	40,489,190.2 +/− 38,430,192.9 (*10 tests*)	179,909.38 +/− 214,954.87	54.0 +/− 0.0
Difficult	Timeout	317,279.8 +/− 194,788.25 (*10 tests*)	92.1 +/− 0.94
Evil	Timeout	1,322,176.6 +/− 1,372,281.50 (*10 tests*)	200.9 +/− 87.68

Table 3. Backtracking Search Algorithm (without randomization of variable and value orders.

	Time (ms)	**# Nodes**
Easy	4.26 +/− 3.88	1570.0 +/− 0.0
Medium	7.92 +/− 4.84	3356.0 +/− 0.0
Difficult	16.9 +/− 9.38	12808.0 +/− 0.0
Evil	28.06 +/− 12.51	18035.0 +/− 0.0

2d. Discussion of the results

I started with an implementation of a simple **backtracking search algorithm**, which is essentially a depth first search algorithm with two modifications. Firstly, the algorithm only considers assignments to a single variable at each step. Secondly, it only considers values that don't conflict with previous assignments. These modifications make sure that the algorithm takes into consideration **commutativity**, a crucial property common to all CSPs.

In the question it was asked to pick and assign variables and values in a random order. By comparing these results (Tables 2 and 3) with the case when the variables and values were assigned in some order (Table 3), it can be concluded that the randomness increased time and the number of explored nodes by many orders of magnitude. The reason is that selecting some (non-random) order is equivalent to

using some heuristic, which might be not the best one, but it still performs better than a random order. In the case of the ordered assignments of variables and values, all puzzles were solved in less then 100ms. Whereas in the case of random assignments, it took the algorithm around 1000ms to solve the "easy" puzzle, 40, 500ms to solve the "medium" puzzle. The "hard" and the "evel" puzzles weren't solved within 1,000,000 ms.

Adding **forward checking** to the backtracking search algorithm helped to keep track of remaining legal values for unassigned variables, and the search was terminated when any variable had no legal values. Again, both variables and values were picked and assigned in a random order. As it can be seen from Tables 1 and 2, forward checking improved performance of the back search algorithm in terms of both the time and the number of expanded nodes. The variance for this algorithm was still pretty high predominantly due to the randomness. Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures.

In order to further improve performance of the backtracking search algorithm three heuristics were implemented in addition to forward checking. **Minimum-remaining-values heuristic** implies choosing the variables with the fewest legal values. If some variable has no legal values left, MRV heuristic helps to detect the failure earlier, whereby avoiding unnecessary search through other variables. In a situation when several variables had equal number of legal values, the **degree heuristic** was used as a tiebreaker. This heuristic implies selecting the variable with the largest number of constraints on other unassigned variables. It helps to reduce branching factor. The order of assigning values was governed by the **least-constraining-value heuristic,** which prefers a value that rules out the least number of choices for neighboring unassigned variables. If there were more ties left in the order of variables or values, they were resolved randomly. After applying these heuristics the performance significantly improved both in terms of the time and in terms of the number of expanded nodes. Variance also substantially decreased due to the decrease of randomness.