Sudoku as a Constraint Satisfaction Problem

Formal Definition

Sudoku can be defined as a Constraint Satisfaction Problem with:

- X – The set of variables
- D – The set of domains of each variable
- C – The set of constraints for each variable

For the case of Sudoku, these are as follows:

- X – The set of 81 variables named '1', '2', '3'……upto '81'. The variable named X_i represents the X_ith cell of the Sudoku
- D – where each D_i is defined as
  - D_i = {1, 2, 3, 4, 5, 6, 7, 8, 9} for an empty cell
  - D_i = {assignedValue} – The domain of an assigned cell only has a single value, which is the assigned value
- C – The set of 27 *AllDiff* constraints, which can be informally defined as:
  - One for each row - AllDiff(1,2,3,4,5,6,7,8, 9) for 1st row, AllDiff(10, 11, 12, 13, 14, 15, 16, 17, 18) for 2nd row etc upto 9th row
  - One for each column – AllDiff(1,10,19,28,37,46,55,64,73) for 1st column, AllDiff(2,11,20,29,38,47,56,65,74) for 2nd column etc upto 9th column
  - One for each box (3X3 square) – AllDiff(1,2,3,10,11,12,19,20,21) for 1st box, AllDiff(4,5,6,13,14,15,22,23,24) for 2nd box etc upto 9th box

  Where *AllDiff*(X1, X2, X3, X4,X5,X6,X7,X8,X9) is True if all the values of all the variables are different, otherwise False

With this definition, a **state** of the Sudoku CSP is any assignment (partial or complete) of the 81 variables. And a **solution** of the the Sudoku CSP is any complete assignment of all the 81 variables satisfying all the 27 *AllDiff* constraints

Results

Version A: Standard Backtracking Search

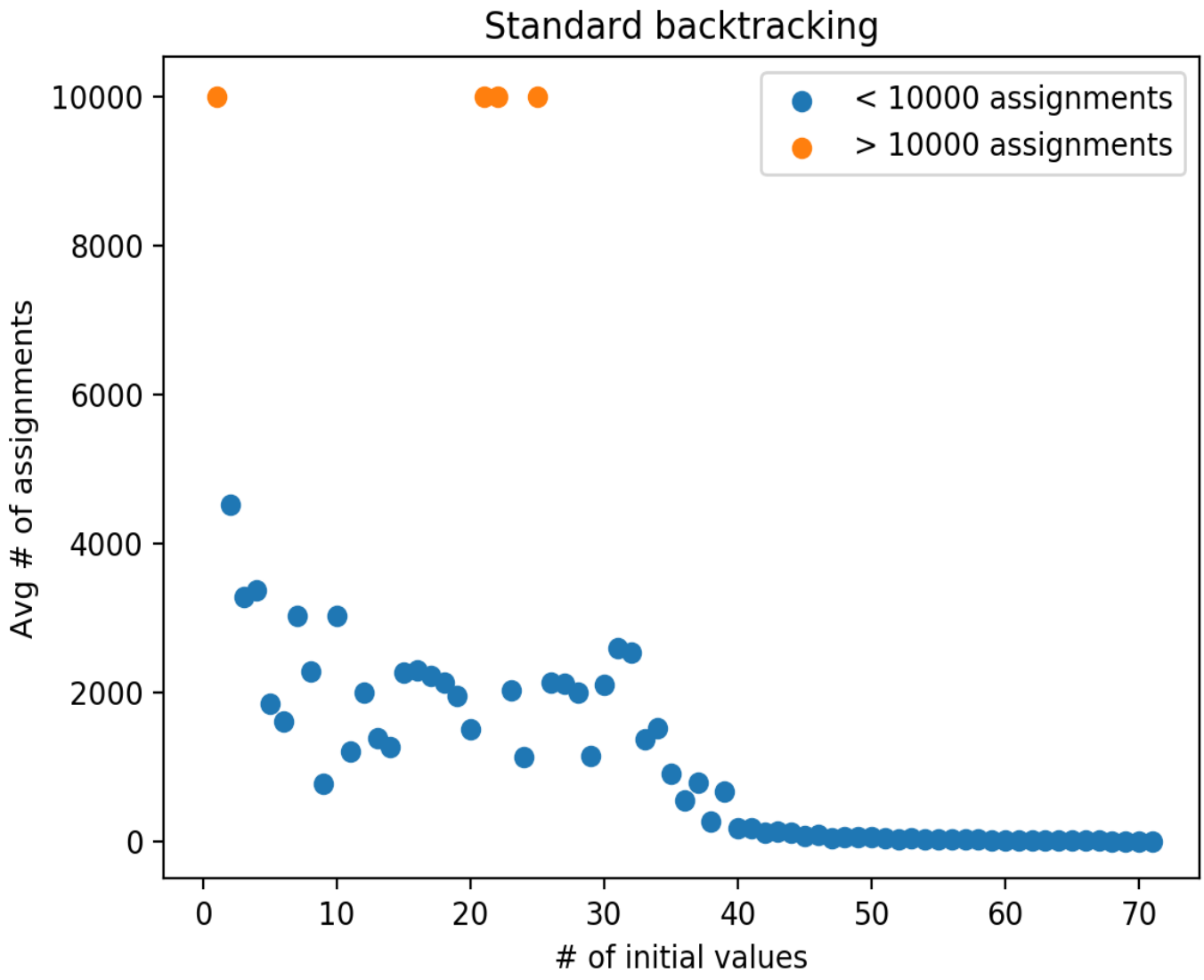

Figure 1

- Out of 710 instances, 90 instances crossed 10, 000 assignments
  - For 4 data points (where each data point Is the average of 10 sudoku problems), 10, 000 steps was crossed majority of the times. This is represented by the 4 orange dots

Version B: Standard Backtracking Search with Forward Checking

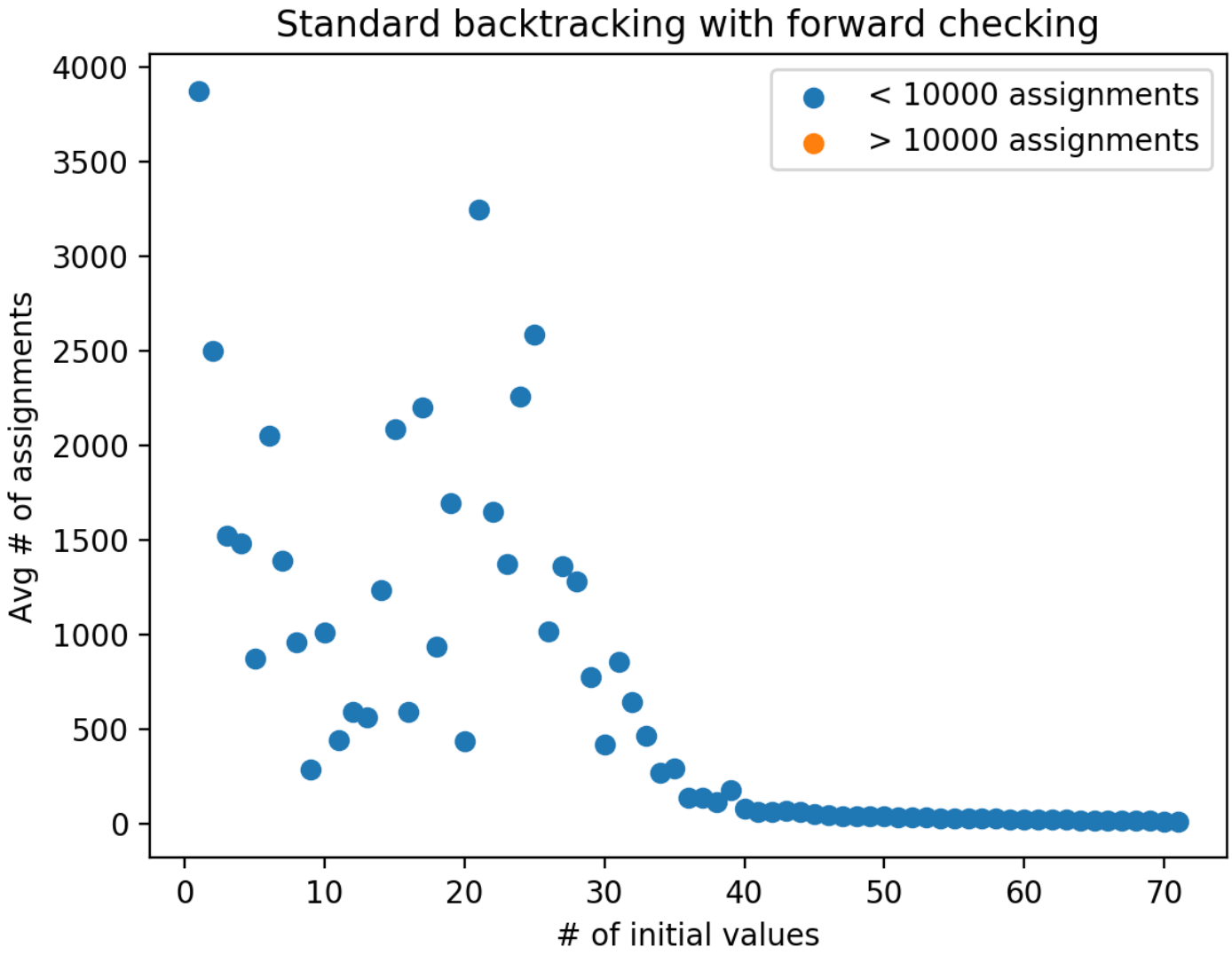

Figure 2

- Out of 710 instances, 43 instances crossed 10, 000 assignments

Version C: Standard Backtracking Search with Forward Checking and Heuristics (MRV, MCV and LCV)

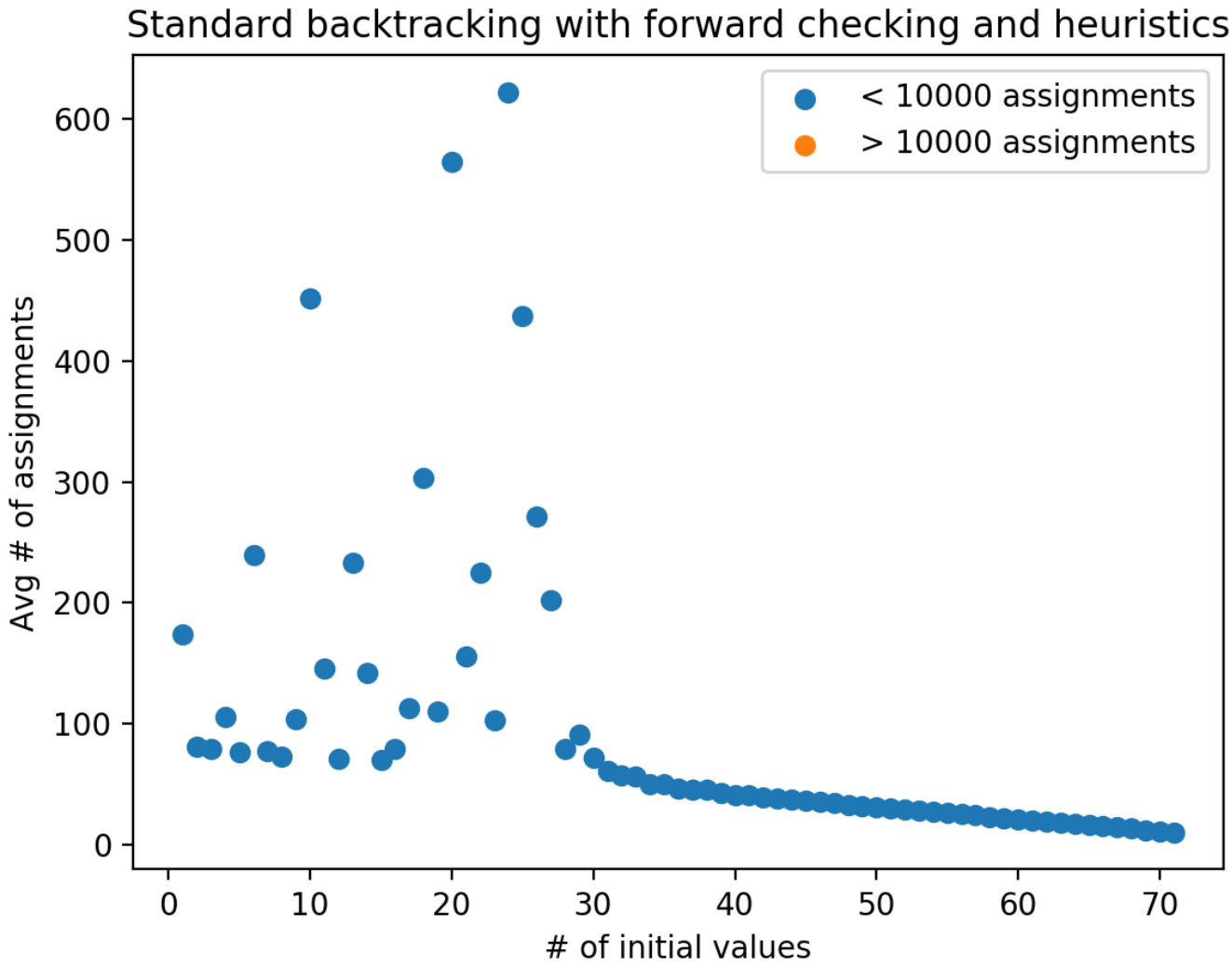

Figure 3

- Out of 710 instances, 14 instances crossed 10, 000 assignments

General Observations across all the 3 versions:

The entire state space, including solution Sudokus and incomplete Sudokus, is incredibly huge. As calculated by Bertram Felgenhauer and Frazer Jarvis in 2005, the number of solution sudokus alone is 6,670,903,752,021,072,936,960.

- The number of times, the algorithm crosses 10, 000 steps is decreasing as we go from version A to B to C
- For any given version (A or B or C), the Average # of assignments (y-axis value) is on a decreasing trend in general. This means that greater the

number of initial values, the easier is the Sudoku and the algorithm solves it faster. But, there are cases where Average # of assignment increases, even though # of initial value increases, i.e y2 > y1 when x2 > x1. This is because, the count of initial values alone does not help in solving the Sudoku. How the initial values are distributed (whether they are spread across the Sudoku, whether they are same numbers or different numbers) plays a significant role.
For example, consider the below 2 Sudokus both with 17 initial values:

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

Figure 4: Version C solved it in 196 assignments

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

Figure 5: Version C took > 10, 000 assignments

- In terms of performance, Version C > Version B > Version A as shown by the plots and the below table
  - 2 exceptions to this are marked in red in the below table, where Version A performed better than Version B

# of initial Values	Version A (Avg # of assignments)	Version B (Avg # of assignments)	Version C (Avg # of assignments)
1	>10000	3872	174
2	4523	2499	80
3	3298	1522	79
4	3376	1482	105
5	1861	870	76
6	1621	2048	239
7	3044	1390	76
8	2284	959	73
9	782	286	103
10	3041	1007	452
11	1213	437	145
12	2000	588	71
13	1395	561	233
14	1271	1235	142
15	2273	2086	69
16	2304	590	78
17	2232	2196	112
18	2143	933	303
19	1967	1693	110
20	1518	432	565
21	>10000	3245	155
22	>10000	1649	225
23	2036	1369	102
24	1137	2255	622
25	>10000	2582	437
26	2146	1014	271
27	2124	1357	201
28	2011	1281	79
29	1156	772	91
30	2112	418	71
31	2607	851	60
32	2547	641	57
33	1378	462	56
34	1531	270	50
35	915	289	49
36	563	136	45
37	793	136	45

38	275	113	45
39	683	177	42
40	186	76	41
41	192	60	41
42	119	58	39
43	139	63	38
44	122	59	37
45	78	48	36
46	94	44	35
47	55	38	34
48	69	37	33
49	64	37	32
50	61	37	31
51	52	34	30
52	41	31	29
53	53	31	28
54	34	28	27
55	40	27	26
56	33	26	25
57	30	25	24
58	31	24	23
59	25	22	22
60	24	21	21
61	25	20	20
62	22	19	19
63	19	18	18
64	19	17	17
65	17	16	16
66	16	15	15
67	15	14	14
68	13	13	13
69	12	12	12
70	11	11	11
71	10	10	10

- When the # of initial values approaches closer to 81, the performance of all 3 versions become same as the Sudoku gets easier to solve
    - Forward Checking has no values to eliminate from the domains of the neighboring cells(same row, same column and same box)
    - Ordering of the variables and their values don't work anymore because there are fewer remaining variables with fewer possible values

Implementation Details

- We have to take the input data and set the Di's accordingly
- Choosing the Most Constrained variable –
    - Choosing the variable with smallest domain
    - How do we choose the most constraining variable (for tie breaking) ?
        - Choose the variable (i.e cell) with maximum degree – i.e the cell with the most number of unassigned cells that it can constrain
            - Need a method to find if a cell is assigned or not
            - Need a method that returns all neighbors of a cell
    - Least Constraining value
        - While choosing the value of a particular cell, choose that value that rules out fewest values for its peers
            - Need a method that takes (variable, value) – and returns the updated domains of all its neighbors
                - This method can be used for Forward checking
            - Using the above method we can find the value assignment that caused the least no of reductions in domains of its neighbors
- Variable assignments needs to be counted
    - CSP can be a class with a counter for this inside it
- Need a method for evaluating the AllDiff constraints on the 9 variables
    - For each variable, there will be 3 AllDiff evaluations – row, column and diagonal
        - So we need a method that returns the participating variables in the AllDiff constraint
- We need to pass a copy of the Sudoku to each recursive call, so that while backtracking we don't have issues
- We need a converter for converting list of variables into a Sudoku grid, and viceversa
    - Let there be a SudokuGrid class for that
- Do we need a constraint graph

Question

Version 1

- Make the algorithm generic so that version2 and version3 can be easily plugged in to version 1

Online resources

- https://www.sudoku-solutions.com/
- http://www.sudokuwiki.org/sudoku.htm

Running Processes
21507 – shreesha computer – (v2, v3) from 1 to 35
24431 – shreesha computer - (v2, v3) from 36 to 71
3194 – my first computer – v1 from 1 to 35
14834 – igor computer – v1 from 36 to 71