Preamble: You, and How to Run Your Code

My name, ID#, UCInetID: Jason Chen-Ju Hsieh, 84655438, jchsieh1@uci.edu
Partner name, ID#, UCInetID: Haoming Huang, 63522953, haomingh@uci.edu

By turning in this assignment, I/We do affirm that we did not copy any code, text, or data except CS-171 course material provided by the textbook, class website, or Teaching Staff.

The programming language(s) and versions you used in your project:
Java 1.7
The environment needed to compile and run your project:
N/A

A small write up of your implementation.

Since we are using the provided java shell, the majority of the program's structure has already been determined. The project files are organized as such: the Sudoku folder contains files that define a Sudoku puzzle and all of its components; the cspsolver folder contains files that specify components necessary in solving Sudoku puzzles, such as constraints, domains, and variables; the main folder contains the main program.

The program takes arguments that specify input file, which represents a Sudoku puzzles, an output file, a timeout to stop the puzzle solving process if it takes too long, and several tokens which represents tools to be used to solve the puzzle. The main program takes these arguments and attempts to solve the puzzle accordingly.

BTSolver.java contains all of the tools necessary to solve a Sudoku puzzle which includes:

- Constraint checks (Forward checking (FC) , EXTRA CREDIT: maintain arc consistency (MAC), Arc consistency preprocessing (ACP))
- Variable selection heuristics (Minimum remain values (MRV), Degree Heuristics (DH))
- Value selection heuristics (Least constraining value (LCV))

The output describes the outcome of solving the puzzle, the time it took to solve the puzzle, and the node counts.

Part 1: (Required) What You or Your Team Did

[Fill in the check box below to indicate what you or your team did. Put exactly one "X" in each triplet of Yes/Partly/No for each line item. Each Yes/Partly/No triplet must have exactly one "X".

I/We coded it.			**I/We tested it thoroughly.**			**It ran reliably and correctly.**			**What was it?**
Yes	Partly	No	Yes	Partly	No	Yes	Partly	No	
Required Coding Project									
		X	X			X			Backtracking Search (BT)
X			X			X			Forward Checking (FC)
X			X			X			Minimum Remaining Values (MRV)
X			X			X			Degree Heuristic (DH)
X			X			X			Least Constraining Value (LCV)
Extra Credit									
		X			X			X	Writing Your Own Shell
		X			X			X	Writing Your Own Random Problem Generator
X			X			X			Arc Consistency AC-3/ACP/MAC
		X			X			X	Local Search using Min-Conflicts Heuristic
X			X			X			Advanced Techniques, Extra Effort, or Creativity Not Reflected Above (*)

(*) Advanced Techniques, Extra Effort, or Creativity Not Reflected Above:

We created automated test programs for Parts 2 to 5 that auto generates results for the tables. (See appendix I & II)
Note: if a testing program does not count as extra effort then please do not mark any points off, please disregard the last row of the table above and all following descriptions of the test programs, along with the appendixes.

Part 2: N=9 (9x9) Sudoku: Analysis of Best Methods Combination

Fill in the following table based on your system's performance on the "hard" 9x9 problems: PH1, ..., PH5 on the supplied coding shell. For each row in the table, run your system on PH1-5 with the indicated option tokens turned on, then report the average number of nodes expanded, the average time taken, and the standard deviation of the time taken. The blank line at the top is for the case of using backtracking search only; For breaking ties, order the variables lexicographically, and order the values 1 to 9. For the next seven lines, each with only one option token, run your system with only that single token enabled. (ACP, MAP, and Other are optional; if you did not do them, leave them blank.) On the next line, run FC, MRV, DH, and LCV all together. Finally, do experiments with other combinations of methods, and fill in the last blank line to indicate the combination you found to be fastest by putting an "X" under any option token that it uses.

FC	**MRV**	**DH**	**LCV**	**(ACP)**	**(MAP)**	**(OTHER)**	**AVERAGE # NODES**	**AVERAGE TIME**	**STD. DEV. TIME**
The blank row below is for the case of no heuristics and no constraint propagation.									
							1796078	263.8762	72.3336
X							430836	180.4834	146.3927
	X						1841633	263.8496	72.3953
		X					1855005	300.0974	0.05882
			X				1871294	262.237	75.614
				(X)			1521556	261.7208	76.5794
					(X)		469181	176.8066	143.6417
						(X)			
X	**X**	**X**	**X**				79566	35.4104	49.01676
	X	**X**	**X**	**X**	**X**		35593	17.081	20.38483
Fill in the blank row above with the combination you found to be fastest on PH1-5.									

Did you get the results you expected? Why or why not?

Yes, DH by itself cannot complete any puzzles and therefore have an average time of 5 minutes, which is the time limit. FC and MAP were the fastest by themselves due to all the consistency checks. Our own combination of using ACP and MAP instead of FC managed to cut down node count by more than half.

Did you implement any Advanced Techniques, Extra Effort, or Creativity not reflected above? If so, please tell us what you did.

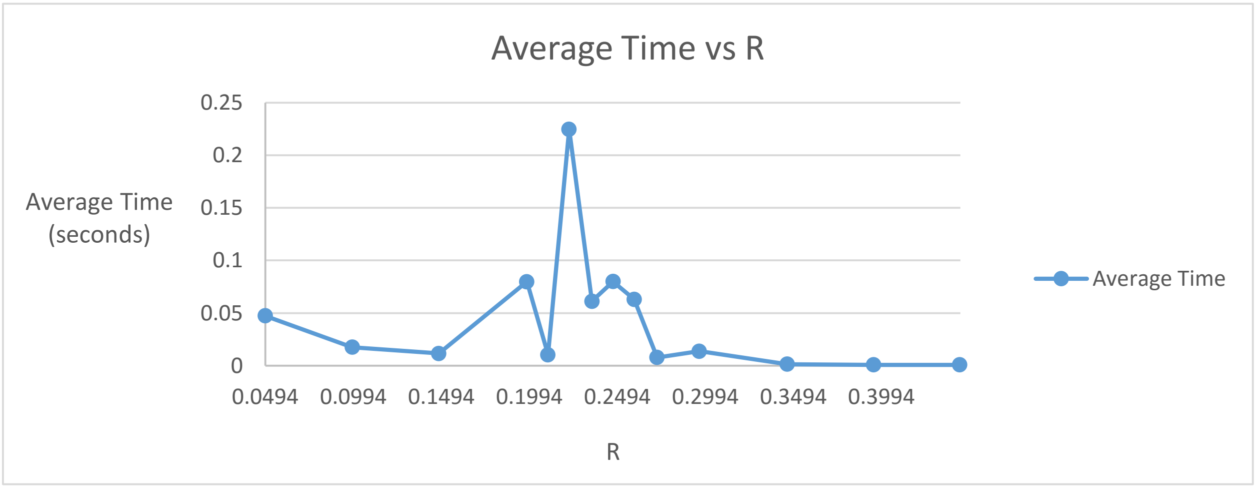
We wrote an automated testing program to obtain these results since running each test would take too long and become difficult to organize. The program uses different tokens in order and outputs the result in a way that can be easily transferred into the table above.
The same program is modified and used for parts 3, 4, and 5 (See appendix I & II)

Part 3. N = 9 Sudoku: Estimate the Critical Value "**Hardest R**"

3.1. Fill in the following table, for total time, where R = M / N^2.
[Average together at least 10 different puzzles for each data point. Feel free to use different data points if it yields a more informative picture of your system running its best combination above.]

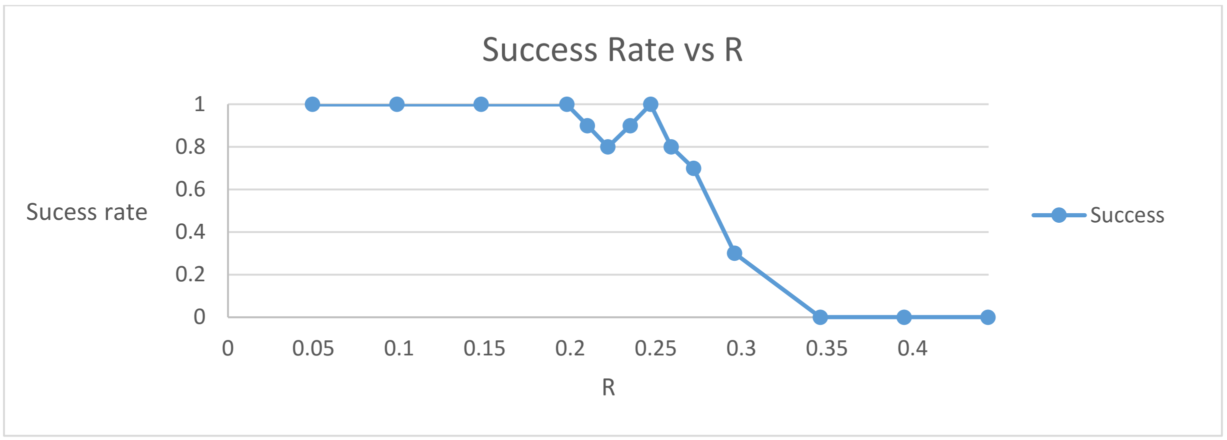
M	N	P	Q	R = M / N^2	AVERAGE # NODES	AVERAGE TIME	STD. DEV. TIME	# (%) SOLVABLE
4	**9**	**3**	**3**	**0.0494**	48	0.0473000	0.0779141	1.00
8	**9**	**3**	**3**	**0.0988**	42	0.0175000	0.0106981	1.00
12	**9**	**3**	**3**	**0.148**	36	0.0116000	0.0019596	1.00
16	**9**	**3**	**3**	**0.198**	381	0.0798000	0.2120787	1.00
17	**9**	**3**	**3**	**0.210**	40	0.0104000	0.0086046	0.90
18	**9**	**3**	**3**	**0.222**	1182	0.2248000	0.6095275	0.80
19	**9**	**3**	**3**	**0.235**	337	0.0612000	0.1329134	0.90
20	**9**	**3**	**3**	**0.247**	505	0.0799000	0.1207265	1.00
21	**9**	**3**	**3**	**0.259**	331	0.0630000	0.1637327	0.80
22	**9**	**3**	**3**	**0.272**	30	0.0078000	0.0053254	0.70
24	**9**	**3**	**3**	**0.296**	71	0.0137000	0.0172688	0.30
28	**9**	**3**	**3**	**0.346**	3	0.0014000	0.0009165	0.00
32	**9**	**3**	**3**	**0.395**	2	0.0007000	0.0004583	0.00
36	**9**	**3**	**3**	**0.444**	2	0.0007000	0.0004583	0.00

3.2. Find the critical value of the "hardest R" for N = 9 and your best combination above.
[Based on your data in 3.1 above, produce a graph similar to that shown below.]



Based upon your results above, you estimate the "hardest R$_9$" = <u>.222</u>

3.3. How does puzzle solvability for your best combination vary with R = M / N^2?
[Based on your data in 3.1 above, produce a graph similar to that shown below.]



3.3 Is the critical value for "hardest R" approximately the same as the value of R for which a random puzzle is solvable with probability 0.5?

Somewhat, however the hardest R clearly and consistently takes the longest time to solve when compared to other values of R.

4. What is the "**<u>Largest N</u>**" You Can Complete at the "**<u>Hardest R_9</u>**"?

Fill in the following table using your best combination of methods from Part 2 above. Fix a time limit of **<u>5 minutes or less</u>** to complete each new puzzle. For each new value of N, scale M with the R_9 you found in Part 3 as:

Scaled M = round ($N^2 \times R_9$)

Consider at least 10 random puzzles for each new value of N. Report the number and percentage of random puzzles that your best combination was able to complete for that N in **<u>5 minutes or less.</u>** For those puzzles that your system completed, report average nodes, average runtime, and standard deviation of the runtime. (Standard deviation is "none" if your system completed only one puzzle.) When your system fails to complete any puzzles for some value of N, it is unnecessary to continue.

"Hardest M" round ($N^2 \times R_9$)	**N**	**P**	**Q**	**# (%) Completed in 5 Minutes or Less**	**AVERAGE # NODES (Completed puzzles only)**	**AVERAGE TIME (Completed puzzles only)**	**STD. DEV. TIME (Completed puzzles only)**
32	12	3	4	1	279	0.081	0.176837
50	15	3	5	0.6	32658	9.18	19.85268
57	16	4	4	0.5	62171	31.2748	49.39893
72	18	3	6	0.8	951	0.71825	1.000626
89	20	4	5	0.4	59491	60.76375	63.69176
98	21	3	7	0.3	1275	2.083333	1.540888
128	24	4	6	0.1	308	0.612	0
162	27	3	9	0.1	15385	38.722	0
174	28	4	7	0	0	0	0
200	30	5	6	0	0	0	0
227	32	4	8	0	0	0	0
272	35	5	7	0	0	0	0

Largest N = 12

In some rare situations the best combination is not able to complete all puzzles for N=32, but for N = 15 the percentage of puzzles completed is consistently below 1.0

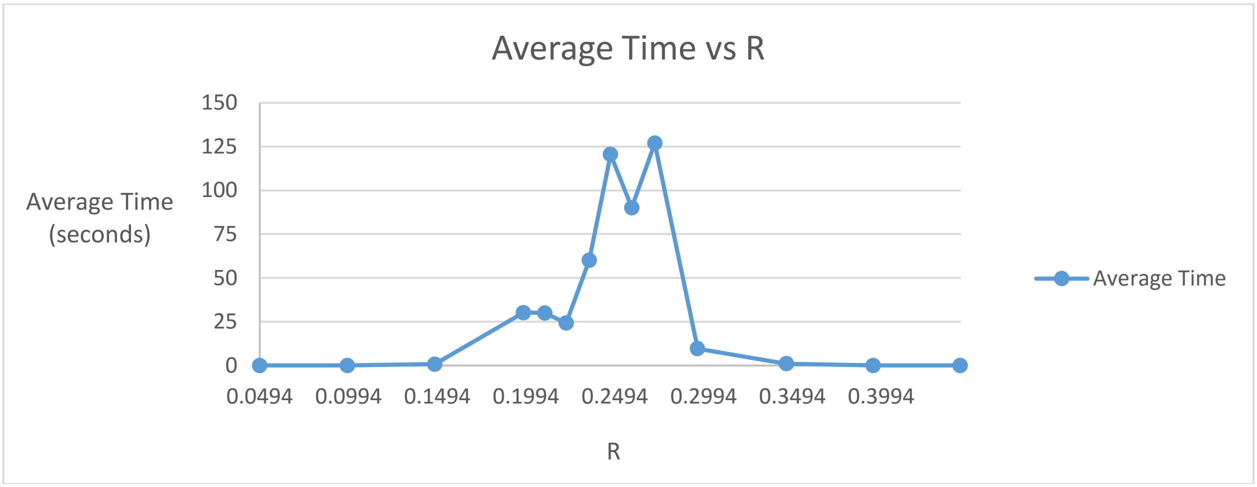
Part 5. Monster Sudoku: Is “Hardest R” constant as you scale up?

5.1. Fill in the following table.

[Fill in N, P, and Q from the largest value of N in Part 4 for which you **completed all puzzles in 5 minutes or less**. Then, try different M values that are computed from R values tried in Part 3. For example, new M values can be round (0.0494 * N_largest^2), round (0.0988 * N_largest^2), etc. Note that 0.0494, 0.0988, are came from the first two rows in table in Part 3. The actual values of M that you use will result in slightly different values of R because of rounding, so recalculate R below for the values of M you actually used. Average together at least 10 different puzzles for each data point. Feel free to use different data points if it yields a more informative picture of your system running its best combination above.]

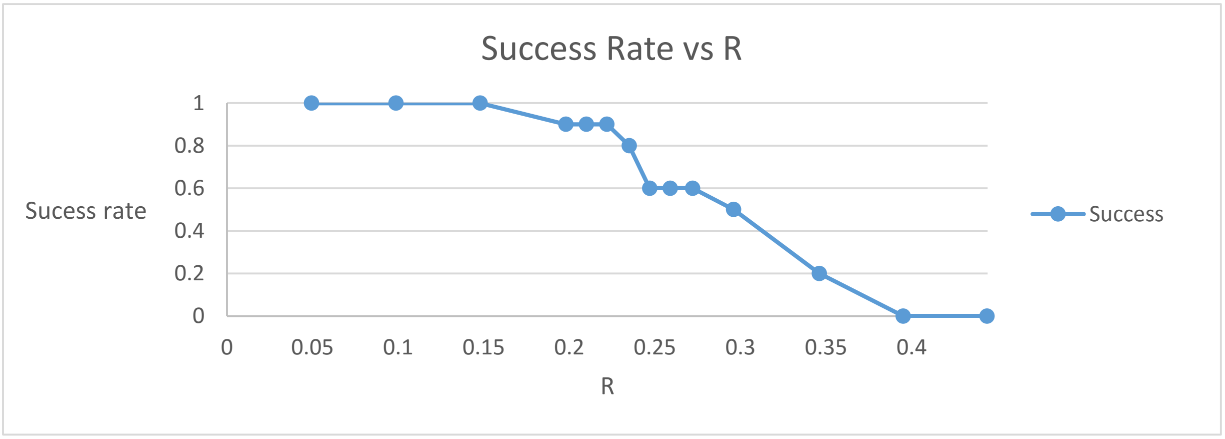
M	N	P	Q	R = M / N^2	AVERAGE # NODES	AVERAGE TIME	STD. DEV. TIME	# (%) SOLVABLE
7	12	3	4	0.0494	94	0.044	0.0184174	1
14	12	3	4	0.0988	82	0.0391	0.0146045	1
21	12	3	4	0.148	2864	0.7197	2.0901368	1
29	12	3	4	0.198	147630	30.2051	89.9387276	0.9
30	12	3	4	0.21	145674	30.0131	89.9989667	0.9
32	12	3	4	0.222	120618	24.2696	48.7196321	0.9
34	12	3	4	0.235	270344	60.0321	119.9887032	0.8
36	12	3	4	0.247	592160	120.5389	146.5390794	0.6
37	12	3	4	0.259	417208	90.093	137.4213995	0.6
39	12	3	4	0.272	573628	126.8781	142.7322099	0.6
43	12	3	4	0.296	47255	9.5821	22.1066708	0.5
50	12	3	4	0.346	4956	0.9849	2.4378494	0.2
57	12	3	4	0.395	2	0.0008	0.0006	0
64	12	3	4	0.444	2	0.0005	0.0005	0

5.2. Find the critical value of the "hardest R" for N_largest.
[Based on your data in 5.1 above, produce a graph similar to that shown below.]



Based upon your results above, you estimate the "hardest $R_{N\_Largest}$" = <u>.272</u>

5.3. How does puzzle solvability for your best combination vary with R = M / N^2?
[Based on your data in 5.1 above, produce a graph similar to that shown below.]



5.4 Is the critical value for "hardest R" approximately the same as the value of R for which a random puzzle is solvable with probability 0.5?

Yes, very close

5.5 <u>Research Question:</u> Is <u>hardest $R_{largest\_N}$</u> ≈ <u>hardest R_9</u>? Why or why not? How is the hardest R related to the board parameter N? What is the shape of the function R(N), which gives you the hardest ratio for 9, 12, 15, 16, 18, 20, …, 35?

Yes, they are reasonably close given that R could range from 0 to 1.
For hardest R_9, there should be a number m that's just right for the puzzle to take the longest time to complete, this is the same case for hardest R_{12}. In both cases we see that the hardest R is clearly somewhere in the middle, where M is not too large or too small, as supported by the graphs. This suggests that there's a constant ratio between the number of initially filled in cells (M) and the number of cells on the board (N^2)
If we use the same M that gives the hardest R_9 and uses it for a puzzle with different N, it will not produce the hardest R since the board size has changed while the number of initially filled in cells stayed the same. Now there are more cells but still the same amount of initially filled in cells, in this case the graph would suggest that a solution can be found quicker. This further suggests that the hardest R would stay constant.

The hardest R has a linear relationship with the number of cells on a board = N^2, which means that it has a square relationship with N

The shape of the function R(N) would be a curve, since R is defined by M / N^2.

Appendix I: automated testing code for part 2

```
public class GenerateResultsPart2 {

    public static void main(String[] args) throws InterruptedException
    {
      System.out.println("Average nodes        Average time            Std 
Dev Time          Tokens");
      String[] filenames = {"ExampleSudokuFiles/PH1.txt", 
"ExampleSudokuFiles/PH2.txt", "ExampleSudokuFiles/PH3.txt", 
"ExampleSudokuFiles/PH4.txt", "ExampleSudokuFiles/PH5.txt"};
      String[] tokens = {"", "FC", "MRV", "DH", "LCV", "ACP", "MAC", "FC MRV 
DH LCV", "ACP MAC MRV DH LCV"};
      for (String token: tokens)
      {
            double[] times = new double[filenames.length];
            double[] nodes = new double[filenames.length];

            for (int i = 0; i < filenames.length; i++)
            {
                  SudokuFile sf = SudokuBoardReader.readFile(filenames[i]);
                  BTSolver solver = runsolver(sf, token);
                  times[i] = solver.getTimeTaken()/1000.0;
                  nodes[i] = solver.getNumAssignments();
            }

            Statistics stat1 = new Statistics(nodes);
            Statistics stat2 = new Statistics(times);

            System.out.format("%-10d %22.7f %23.7f \t\t%s%n", 
Math.round(stat1.getMean()), stat2.getMean(), stat2.getStdDev(), token);
      }
    }
```

Appendix II: automated testing code for part 3, also used in part 4 & 5 after some modifications

```
public static void main(String[] args) throws InterruptedException
    {
      int[] m = {4,8,12,16,17,18,19,20,21,22,24,28,32,36};
      for (int i = 0; i < m.length; i++)
      {
            double[] times = new double[10];
            double[] nodes = new double[10];
            double[] solvable = new double[10];
            for (int j = 0; j < 10; j++)
            {
                  SudokuFile sf = SudokuBoardGenerator.generateBoard(9, 3, 
3, m[i]);
                  BTSolver solver = runsolver(sf);
                  times[j] = solver.getTimeTaken()/1000.0;
                  nodes[j] = solver.getNumAssignments();
                  solvable[j] = solver.getStatus() == "success" ? 1 : 0;
                  Thread.sleep(1000);
            }

            Statistics stat1 = new Statistics(nodes);
            Statistics stat2 = new Statistics(times);
            Statistics stat3 = new Statistics(solvable);

            System.out.println("Average nodes=" + stat1.getMean()+ "
    Average time=" + stat2.getMean() + "  Average STD time=" + 
stat2.getStdDev() + "   solvable=" + stat3.getMean());
      }
    }
```