My name, ID#, UCInetID: Deon Zhao, 24642298, chunqiz@uci.edu
Partner name, ID#, UCInetID (or "none"): Nanqing Li, 45504495, nanqingl@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: Python 3.4.3
The environment needed to compile and run your project:
N/A

A small write up of your implementation.
We have designed and created 3 different files for submission.
Main.py is the entirety of the required project, and also our self written shell. It accepts all the required inputs and tokens for different ways to solve sudoku problems, and will output a log file as specified in the requirements document.
Generator.py is for extra credit, it accepts an input file name and a output file name and outputs a file based on the ouput file name. All the input file and output file are exactly to require specs.
Ui.py is for creativity extra credit. It is a tool that helped us a lot in debugging the code, as it can run alongside main.py while it is solving to see the sudoku board change dynamically.

Our main.py was designed so that everything is in 1 file for easy Ctrl-F and readability, as splitting everything into multiple files is more confusing and prone to bugs. The shell is completely made from scratch by ourselves and it does everything that is required and needed for this project. The timer and and ui data transfering code is all written with multi threading, so as to not interfere with the main solving loop and drastically hinder solve speed.

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

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:

Clarification on Random Problem Generator extra credit:

The random generator was written in a separate python file called generator.py, it also works by passing it command line arguments: the input filename, and output filename.

Clarification on Arc consistency extra credit:

Arc consistency as in preprocessing, and maintaining it was implemented as part of the Forward Checking solve loop.

Extra Effort/Creativity for extra credit:

A tkinter GUI was implemented to dynamically display the sudoku board, simply run ui.py alongside main.py while it is running to see a dynamically updated sudoku board.

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

All tests below were ran with a timeout of 1 hour.

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.									
							424,872,238	**2287.6 s**	**1619.7 s**
X							**2,006,253**	**2288.6 s**	**1615.7 s**
	X						**484,187**	**441 s**	**735.7 s**
		X					**163,754**	**3600 s**	**0 s**
			X				**1,324,231**	**2882 s**	**1436 s**
				(X)					
					(X)				
						(X)			
X	**X**	**X**	**X**				**159,223**	**228.4 s**	**281.6 s**
X	**X**		**X**						
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?

The results were mostly as expected, back tracking search was supposed to be very slow which was exactly the case. Forward checking however was unexpectedly slow on this problems, although visiting a lot less nodes, the implementation of a domain dictionary hindered its performance. DH as explained by the professor, should also be inherently slow ran by itself, or even with others since it is O(n3), and the results matched by timing out all trials, while only visiting minimal amount of nodes. Everything else is pretty much expected. Some methods are very fast at solving some problems while other methods and good at solving other problems.

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

None.

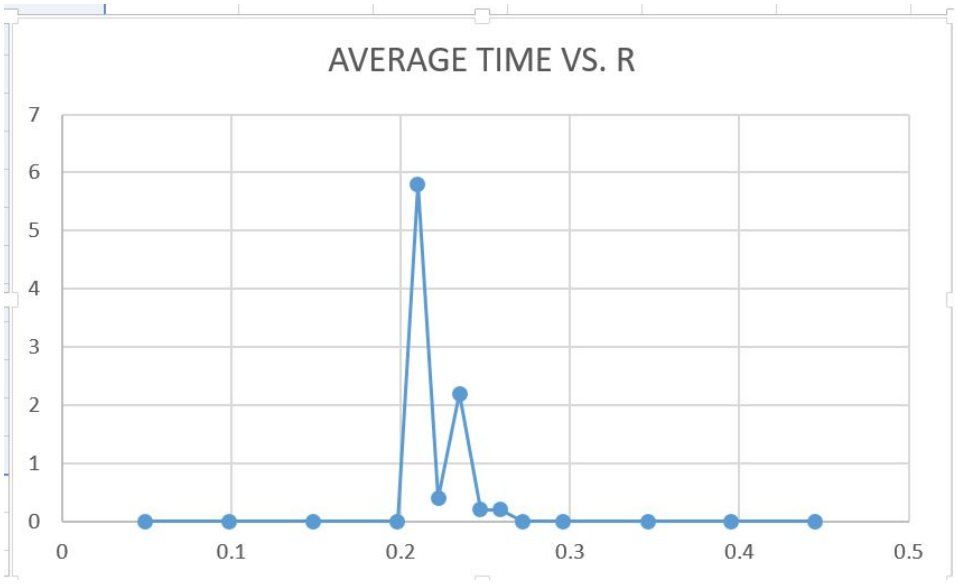
Part 3. N = 9 Sudoku: Estimate the Critical Value "Hardest R"

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.]

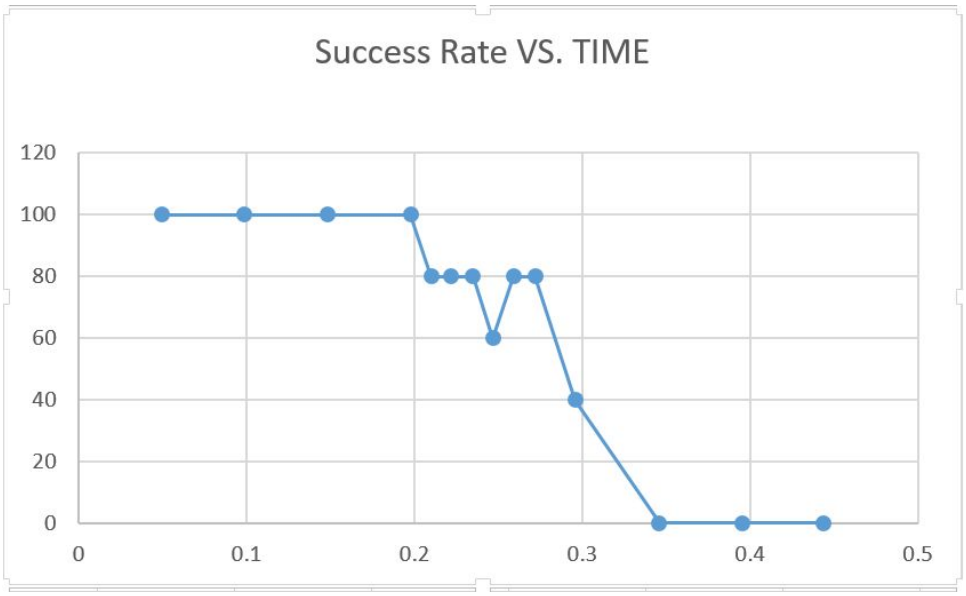
M	N	P	Q	R = M / N^2	AVERAGE # NODES	AVERAGE TIME	STD. DEV. TIME	# (%) SOLVABLE
4	**9**	**3**	**3**	**0.0494**	**77**	**0**	**0**	**100**
8	**9**	**3**	**3**	**0.0988**	**108**	**0**	**0**	**100**
12	**9**	**3**	**3**	**0.148**	**69**	**0**	**0**	**100**
16	**9**	**3**	**3**	**0.198**	**81**	**0**	**0**	**100**
17	**9**	**3**	**3**	**0.210**	6680.6	5.8	134.56	**80**
18	**9**	**3**	**3**	**0.222**	**888.8**	**0.4**	**1.44**	**80**
19	**9**	**3**	**3**	**0.235**	2538.2	2.2	32.32	**80**
20	**9**	**3**	**3**	**0.247**	295.8	0.2	0.16	**60**
21	**9**	**3**	**3**	**0.259**	89.6	0.2	0	**80**
22	**9**	**3**	**3**	**0.272**	**370.8**	**0**	**0**	**80**
24	**9**	**3**	**3**	**0.296**	57.8	**0**	**0**	**40**
28	**9**	**3**	**3**	**0.346**	0.8	**0**	**0**	**0**
32	**9**	**3**	**3**	**0.395**	0.8	**0**	**0**	**0**
36	**9**	**3**	**3**	**0.444**	**0**	**0**	**0**	**0**

3.2. Find the critical value of the "hardest R" for N = 9 and your best combination above.



Based upon your results above, you estimate the "hardest R_9" = 0.21

3.3. How does puzzle solvability for your best combination vary with $R = M / N^2$?



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?

They are somewhat close to each other, but not approximately the same. Considering the few amount of test trials, the statement might be true but more testing data is needed.

4. What is the "Largest N" You Can Complete at the "Hardest R_9"?

Fill in the following table using your best combination of methods from Part 2 above. Fix a time limit of **5 minutes or less** 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 **5 minutes or less.** 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)**
30	12	3	4	100	2272.2	1.8 s	9.76 s
47	15	3	5	60	2832	10.3 s	193.5 s
54	16	4	4	80	324.5	0	0
68	18	3	6	30	1277	3 s	3.5
84	20	4	5	0	0	0	0
	21	3	7				
	24	4	6				
	27	3	9				
	28	4	7				
	30	5	6				
	32	4	8				
	35	5	7				

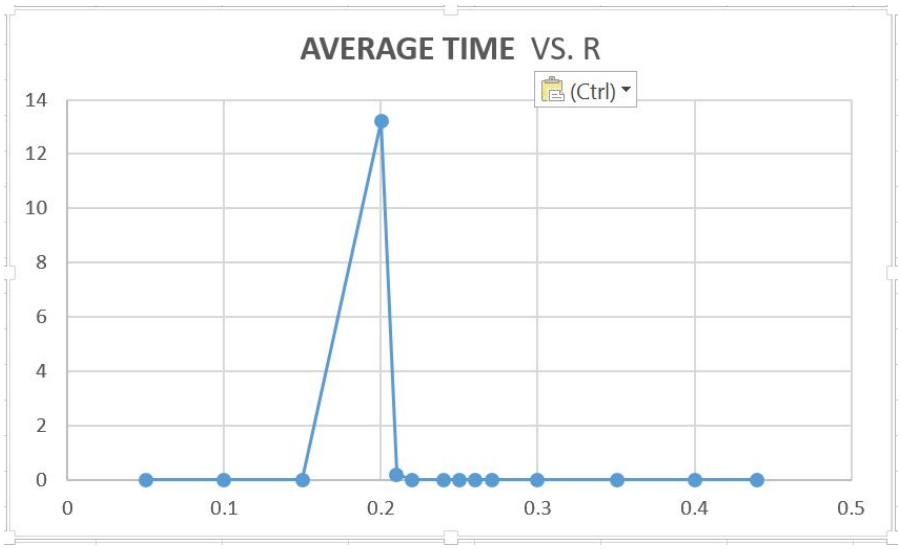
*Got lucky on N=16, only 2 were stuck and timed out, the rest were either instant no solutions, or solved instantly.

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

5.1. Fill in the following table.

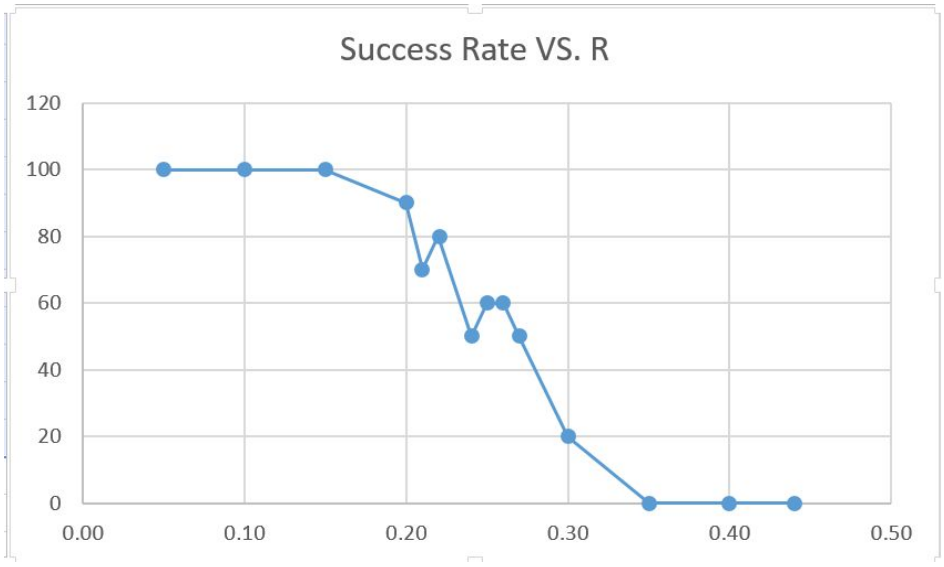
M	N	P	Q	R = M / N^2	AVERAGE # NODES	AVERAGE TIME	STD. DEV. TIME	# (%) SOLVABLE
7	12	3	4	0.05	**137**	**0**	**0**	**100**
14	12	3	4	0.10	**130**	**0**	**0**	**100**
21	12	3	4	0.15	**127**	**0**	**0**	**100**
29	12	3	4	0.20	**8844**	**13.2**	**1568**	**90**
30	12	3	4	0.21	**449**	**0.2**	**0.16**	**70**
32	12	3	4	0.22	**131**	**0**	**0**	**80**
34	12	3	4	0.24	**127**	**0**	**0**	**50**
36	12	3	4	0.25	**140**	**0**	**0**	**60**
37	12	3	4	0.26	**136**	**0**	**0**	**60**
39	12	3	4	0.27	**125**	**0**	**0**	**50**
43	12	3	4	0.30	**49**	**0**	**0**	**20**
50	12	3	4	0.35	**24**	**0**	**0**	**0**
57	12	3	4	0.40	**3**	**0**	**0**	**0**
64	12	3	4	0.44	**0**	**0**	**0**	**0**

5.2. Find the critical value of the "hardest R" for N_largest.



Based upon your results above, you estimate the "hardest $R_{N\_Largest}$" = <u>0.20</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?

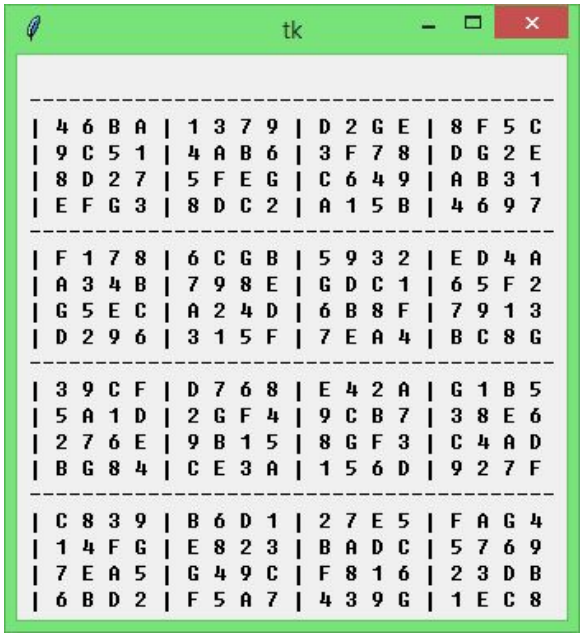
This time it is a little bit more far off, according to the few amount of test data, our R value has decreased slightly to 0.2, and therefore not the same value as R (which should be approximately 0.25) when puzzle is solvable 50% of the time.

5.5 Research Question: Is $hardest\_R_{largest\ N} \approx hardest\_R_9$? 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?

We believe that hardest_R largest N should be approximately equal to hardest R9 in theory. However, due to the nature of our algorithm, our test data has proven otherwise.Our data shows that hardest R decreases as as N increases, hence they have an inverse relationship. The shape of the function R(N) should be a slightly negative sloped line according to our collected data.

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

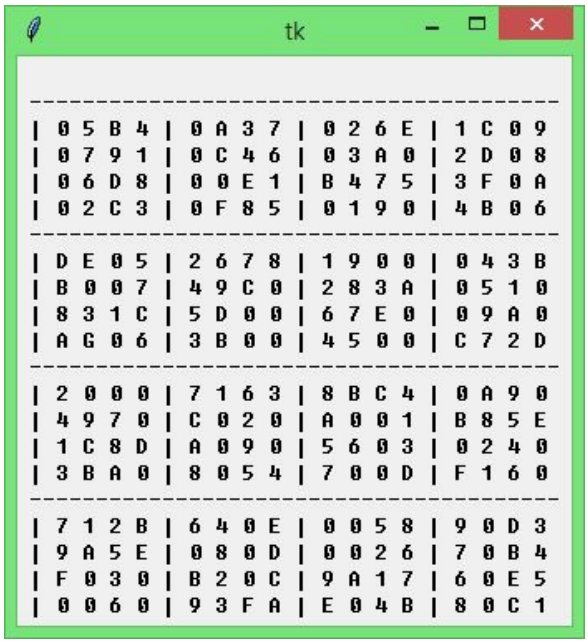
Picture of our dynamic display GUI:



Code for the UI:

```
def updateUI(self):
    txt = ''
    for i, value in enumerate(self.grid):
        if i%self.N == 0:
            txt += '\n'
        if i%(self.N*self.Q) == 0:
            txt += '--' * self.N + '--------\n'
        if i%self.P == 0:
            txt += '| '
        txt += str(value) + ' '
    displayFile = open('display', 'w')
    displayFile.write(txt)
    displayFile.close()
    threading.Timer(1, self.updateUI).start()
```

Picture of it mid solve:



Code for dynamic updating in a thread:

```
import tkinter as tk

UI = tk.Tk()

grid = tk.StringVar()
display = tk.Message(UI, textvariable = grid, font = 'fixedsys')

display.pack()

def update():
    txt = ''
    display = open('display')
    for line in display.readlines():
        txt += str(line)
    grid.set(txt)
    UI.after(1000, update)

update()

UI.mainloop()
```

Ui.py needs to be ran after main.py is run. They also must be located in the same folder/directory. Main.py will call the update function via threading every 1 second, and it will write to a file called ‘display’ within the directory. Then ui.py will read from the display file and update itself with it every 1 second. This has minimal performance impact, as it is only done every second through threading. This feature has helped a lot when debugging because it allows me to see which cells were updated, and if or when the solve loop will finish.