Part 1: You, and How to Run Your Code

Name, ID, UCInetID: Jacob Shoham, 57033266, jshoham@uci.edu
Parter Name, ID, UCInetID: none

By turning in this assignment, I do affirm that I did not copy any text or data except CS171 course material provided to me by the textbook, class website, or teaching staff.

The programming Language(s) I used in this project:

Python

The environment needed to compile and run my project:

Python 2.7 installed on Windows (Not tested on other Operating Systems)

Steps to compile my Generator:

N/A

Steps to compile my Solver:

N/A

Steps to run my Generator:

With the working directory at the top level (CS171Sudoku/), enter the following in the cmd prompt:
> python generator.py <input_file> <output_file>

Steps to run my Solver:

With the working directory at the top level (CS171Sudoku/), enter the following in the cmd prompt:
> python solver.py <input_file>

How to turn on/off the various heuristics and methods:

All user configurable settings can be changed with a text editor in the following file:
</src/settings.py>

This project can also be found at:

https://github.com/jshoham/CS171Sudoku

A small write up of your implementation.

Project Structure:

/CS171Sudoku/ (Top Level)

- /docs/
  - aiproject-monster-sudoku-write-up.pdf
  - report.pdf *(This document)*
- /puzzles/
  - …
  - *(List of Sudoku puzzles from the Internet)*
  - …
- /src/
  - __init__.py
  - generator.py
  - grid.py
  - rw.py
  - settings.py
  - solver.py
  - verifier.py
- /tests/
  - __init__.py
  - grid_tests.py
  - solver_tests.py
  - verifier_tests.py
- /trials n6…/
- … *(trials n6 to n12 contain gathered data on many randomly generated puzzles)*
- /trials n12…/
- solver.py
- generator.py
- move.py *# These four scripts are for running and processing*
- process.py *# bulk timing trials and can be ignored.*
- process_results.py *#*
- trials.py *#*

generator.py:

The generator follows the procedure outlined in the project description, as follows. The generator takes two filename arguments: the first contains the parameters N, p, q, and M specifying the characteristics of generated puzzles, and the second is the file to which output is generated. The procedure to generate a puzzle is to randomly select an empty cell, then assign a random value to it. If that value causes a constraint violation then undo the assignment, eliminate that value as a possible value at that cell, and try again. If a cell has no possible values left, then reset the whole board and restart the whole process. Keep doing this until M cells have been filled. As extra features the generator

can generate and output multiple puzzles at once, and has a timeout feature. Both of these features are detailed later.

grid.py:

The grid is implemented with its main data structure as a 2d array, with each cell in the array containing a field “token” for its value, and another field “possible_values”, which is a set of the possible values that the cell could take. It contains many helper functions to perform actions such as cell assignments, eliminating possible values, performing a forward check, and establishing arc consistency.

rw.py:

This module contains simple helper functions for reading and writing to files.

settings.py:

This module contains all user changeable settings, with comments explaining each setting.

solver.py:

The solver takes an input file containing one or more puzzles, and attempts to solve them sequentially. It does this following the backtracking algorithm as described in Chapter 6 of the textbook (Artificial Intelligence 3rd Edition, Norvig). Additionally the solver has several display and logging features which will be detailed later. A brief outline of the backtracking algorithm is as follows:

1. Choose an empty cell (if there are no empty cells then return the board as it is completed)
2. Order the values in that cell’s domain, then pick the first value to try
3. If the chosen value doesn’t violate a constraint, assign it to that cell (otherwise move to the next value in the ordering)
4. Perform any inferences based on that assignment
5. Perform backtrack() on the resulting board (recursion “magic” happens here!)
6. If backtrack returns a solution then return that solution, otherwise
7. Undo the assignment and inferences
8. Proceed to the next value in the ordering from step 2, then proceed to step 3
9. If there are no more values left in the ordering from step 2 then return None (This step will go "one level up" in the recursion and land at step 6)

verifier.py:

This module contains utility functions to ensure that input conforms to required formats and that all user settings have valid values. If a user setting has an invalid value then it reverts the setting to a default and prints a message announcing the change.

Part 2: What I 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			Sudoku Generator [15]
x			x			x			Sudoku Solver [15]
x			x			x			Backtracking Search [15]
x			x			x			Forward Checking [15]
Extra Credit Bonus Points									
x			x			x			Minimum Remaining Values [5]
x			x			x			Degree Heuristic [5]
x			x			x			Least Constraining Value [5]
x			x			x			Arc Consistency Preprocessing [5]
x			x			x			Arc Consistency after each Step [5]
x			x			x			Monster Sudoku Generator [5]
x			x			x			Monster Sudoku Solver [5]
		x			x			x	Local Search using Min Conflicts [5]

Extra Features:

Overview of extra features:

Generator:

- Multiple puzzle output
- Optional timeout setting if stuck

Solver:

- Forward Checking as a Preprocessing step
- Multiple puzzle input
- Supports additional “inline” format for puzzles (while maintaining full compatibility with project default format)
- Intelligently determines puzzle format of input at run time without needing user intervention
- Optional realtime “animation” of backtracking search
- Configurable logging features

The generator has a few additional features aside from its basic functions. The generator can create multiple puzzles and output them all to the same output file. The output file will contain each puzzle one after another separated by new lines. Note that the solver will accept files containing multiple puzzles in this augmented format. The number of puzzles that the generator will attempt to produce is controlled in “/src/settings.py” under “gen_how_many”.

Additionally, the generator has a timeout setting which can be set in "/src/settings.py" under "gen_time_limit". The timeout behaves similarly to the timeout setting for the solver: if the generator fails to create a board within the time limit it aborts that attempt and continues on the next one if there are more boards to generate. This case usually triggers only when attempting to generate boards with high M values. This feature prevents generator from getting stuck with no foreseeable end.

The solver also has some additional features. I added Forward Checking as a Preprocessing step under "/src/settings.py" under "fcp". When enabled, this process will run a forward check on each initially filled cell at the beginning before search starts.

The solver will accept input containing multiple puzzles without needing any special handling. I also added support for a common "inline" representation of 9x9 puzzles, where each puzzle is represented on a single line as a string of 81 characters. Each cell is represented by the digits 1-9 and blank cells are represented by a dot (".”). For example, an input file containing three puzzles might look like:

52...6.....7.13.....4..8..6.....5.....418.....3..2...87.....
4.....8.5.3.....7.....2.....6.....8.4.....1.....6.3.7.5..2.....1.4.....
48.3.....71.2.....7.5....6....2..8.....1.76...3.....4.....5....

Note however that puzzles in the "inline" format MUST be 9x9 puzzles (This is necessary since each puzzle omits stating its N, p, and q parameters explicitly. While possible, it would require extra handling to accept puzzles with different dimensions). Puzzles using the "inline" format which are not 9x9 will be rejected by the solver as invalid input. The solver will intelligently determine whether the input file is in the regular "default" format or if it is in the "inline" format at run time and proceed accordingly, without needing any special treatment from the user.

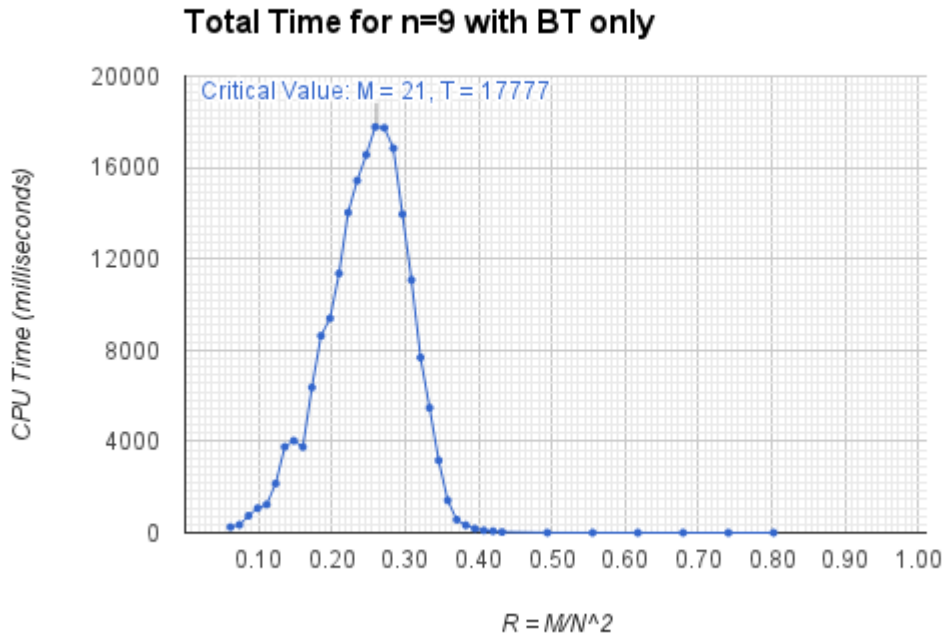
By default the puzzle prints out the current puzzle in progress to the console. It first prints out the original puzzle, followed by the solution (if found) and the timing/assignments/solution/timeout data. In addition to this, there are two optional alternate display settings, realtime and verbose. Realtime treats the console as a frame and "animates" the progress of the solver with each assignment that it makes (Note that while looking awesome this comes at a significant performance cost (up to and exceeding x10-20 slowdown) since a frame is animated with EVERY assignment made). Verbose prints out lots (LOTS) of information about the decision making that is happening during the backtracking search, including things like the next empty cell chosen, the ordering of possible values at that cell, assignments made, assignments undone, and when a cell has run out of possible values. It is highly recommended to only attempt one puzzle at a time if this is turned on, as output can quickly overflow off the top of the console. Both display settings can be controlled in "/src/settings.py" under "solver_display_realtime" and "solver_display_verbose".

The solver has three logging settings which can be turned on and off independently. They are controlled in "/src/settings.py" under the following fields:
"solver_export_solution" - exports puzzles with their solutions if found
"solver_export_raw_data" - exports gathered timing and assignment data in an excel readable format

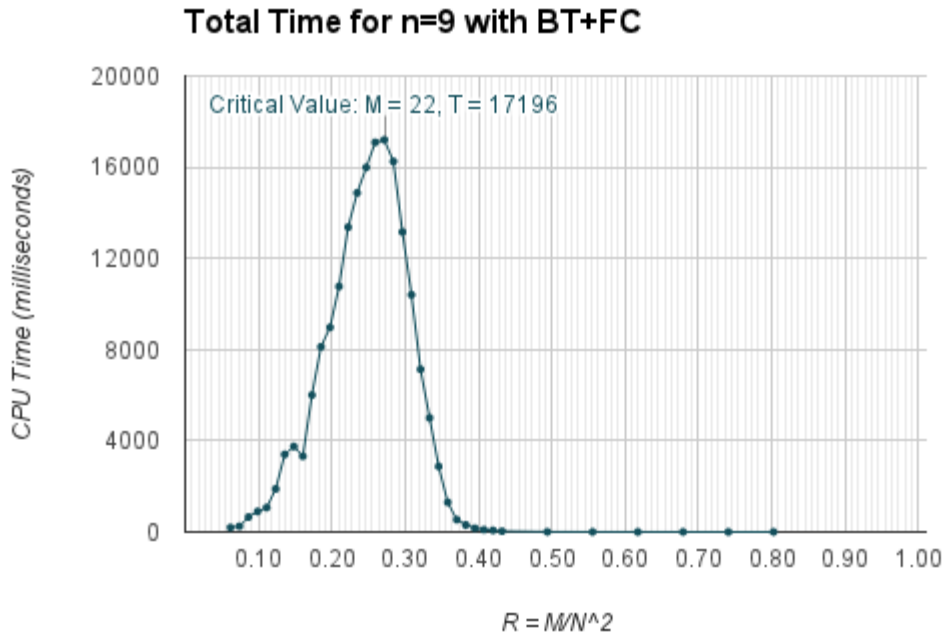
“solver_export_data_summary” - exports a summary of averaged gathered data (This will be pretty much identical to raw_data if the input file contains a single puzzle)
If turned on then each logging setting will write its output to a text file named “<filename>_<tag>.txt” where <filename> is the original input file and <tag> is either “_solution”, “_raw_data”, or “_summary”.

Part 3: Critical Value of "Hardest M" for N = 9

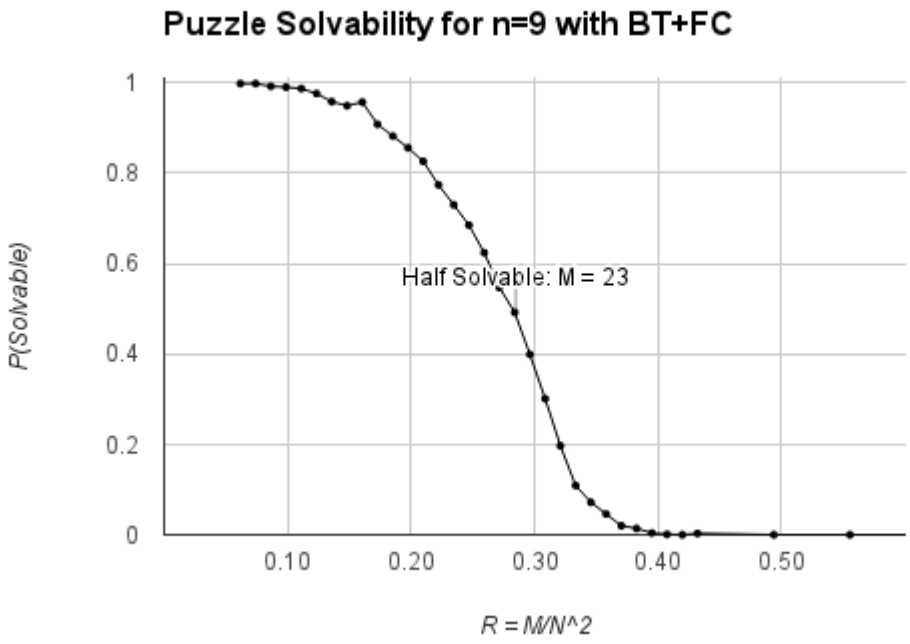
1. **Find the critical value of M for N = 9 with BT Only. Produce a graph of total time vs R = M/N^2.**
   Note: Each data point below represents the average of 1000 randomly generated puzzles.



2. **Find the critical value of M for N = 9 with BT+FC. Produce a graph of total time vs R = M/N^2.**
   Note: Each data point below represents the average of 1000 randomly generated puzzles.



3. **How does puzzle solvability for BT + FC vary with $R = M/N^2$? Produce a graph of P(solvable) vs $R = M/N^2$.**
   Note: Each data point below represents the average of 1000 randomly generated puzzles.



4. **Answer the following:**
   a. **Do you get the same (or approx. the same) critical value of "hardest M" for BT as you did for BT+FC?**
      Yes, BT and BT+FC had nearly identical critical values for "hardest M". The critical value for "hardest M" for BT was M=21 and for BT+FC was M=22. In fact, for BT alone M=21 and M=22 came within 41 milliseconds (0.2%) of each other with total times of 17777 and 17736. For BT+FC the same values of M=21 and M=22 came within 105 milliseconds (0.6%) of each other with total times of 17091 and 17196.
   b. **Do you get the same (or about the same) critical value of "hardest M" using BT+FC for search time only? What is the critical value for search time?**
      The critical value for BT+FC for search time only remains the same at M=22. Initialization time was negligible at 1.13 milliseconds (versus 17195 milliseconds for search time) and thus has no discernable impact on the critical value.
   c. **What percent of "hardest M" puzzles for BT+FC are solvable at all?**
      At M=22, 54.7% of the puzzles were solvable. About 20.1% of puzzles timed out after 60 seconds, meaning just under half of the unsolved puzzles definitely have zero solutions while the other half simply timed out (unknown whether they have a solution or not).
   d. **For what value of M does P(solvable) = ~0.5 occur?**
      For BT+FC at M=23, P(solvable) = 49.2%. (Includes timeouts and zero-solution puzzles)

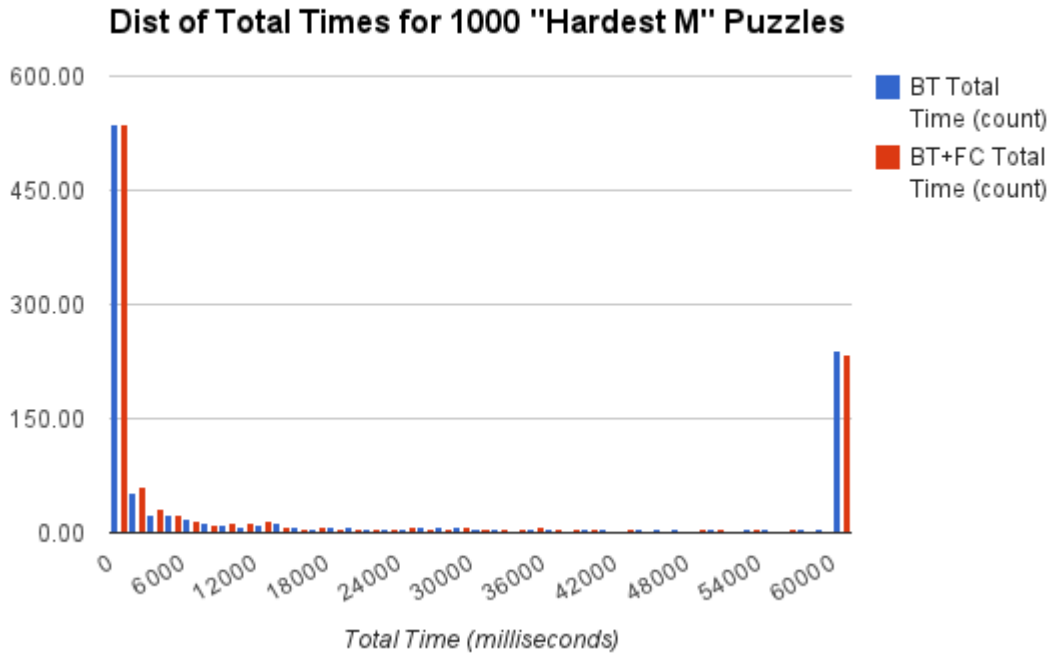
e. **For "hardest M", what is the mean and standard deviation of the total time for BT? For BT+FC? Are those mean times significantly different, statistically?**

For BT, the "hardest M" of 21 has a mean total time of 17777 milliseconds with a whopping standard deviation of 25316 milliseconds.

For BT+FC, the "hardest M" of 22 has a mean total time of 17196 milliseconds with a similarly huge standard deviation of 25045 milliseconds.

These times are so close that they are not significantly different. In general the average performances of BT and BT+FC are nearly the same. However, for any given puzzle BT+FC will be pretty much always marginally faster than BT alone.

The huge standard deviations observed here can be illustrated with the following graph, which shows the distribution of total times for 1000 "hardest M" puzzles. Notice that by far the majority of puzzles either solve very quickly, or reach the timeout limit of 60 seconds.



Part 4: Combinations #3-7 above for N=9

It ran reliably and correctly.			Mean (Sdev) Total Time	Mean (Sdev) Search Time	Hardest M Value	Half-Solvable M Value	What was it?
Yes	Partly	No					
x			23471 (28123)	23471 (27982)	21	22	**BT+FC+MRV**
x			42761 (24660)	42759 (24660)	24	20	**BT+FC+MRV+DH**
x			43790 (23704)	43790 (23824)	24	18	**BT+FC+MRV+DH+LCV**
x			3256 (12783)	3209 (12783)	17	23	**BT+FC+MRV+DH+LCV+ACP**
x			5500 (16116)	5451 (16116)	18	23	**BT+FC+MRV+DH+LCV+ACP+AC**

Answer these questions:

1. **Which more sophisticated methods let you solve N=9 for "hardest M" puzzles faster?**
   ACP was by far the best performer, taking on average 3256 milliseconds to solve its hardest M puzzles. The second best was AC, taking 5500 milliseconds on average.
2. **Which more sophisticated methods have overhead costs that outweigh their potential benefit? (i.e., they make your solver run slower.)**
   Surprisingly, MRV, DH, and LCV all made the solver run slower than BT and BT+FC. There are several circumstances that could have contributed to this unexpected result. First, I believe that my implementation for DH and LCV were poorly executed. More specifically, each invocation of the peers() function creates a new list (allocating memory), slowing down DH and LCV significantly. Running Python's cProfiler tool on my solver indeed showed that a significant fraction of time was spent on the peers() function (A newer implementation of peers() is implemented for this project on a separate branch on Github, but no timing data has been collected). Second, each trial run for the above combinations was done on sets of 100 puzzles (instead of 1000 for BT and BT+FC), so the results here could be reflecting greater random variation from the smaller sample set.
3. **Which more sophisticated methods have mean total run times that are significantly different statistically (up or down; i.e., this is a two-tailed test of significance) from the mean total run time of BT+FC?**
   DH and LCV had mean times that were significantly greater (over 2x slower) than BT+FC. Conversely, ACP and AC had mean times that were significantly faster (4-7x faster) than BT+FC.
4. **Do you get (approximately, i.e., within sampling error) the same values of "hardest M" and "half-solvable M" for the different combinations that you tested?**
   The hardest M values were similar but not quite the same. BT, FC, and MRV seemed to get around the same hardest M of 21-22. Meanwhile the much slower DH and LCV combinations

appeared to converge on a slightly higher hardest M of 24. Finally ACP and AC got the smallest hardest M value of 17-18. The trend seems to be that the 'stronger/faster' combinations tend to have a smaller hardest M value while the 'weaker/slower' combinations have a larger hardest M value.

The figure for half solvable M depends mainly on two aspects: first, the probability that randomly generated M puzzles are actually zero-solution puzzles, and second, the chance that a particular combination will timeout before deciding if a puzzle has a solution or not. The result we get from a timing run can be thought of as approximately "reported half solvability" = "true half solvability" + "timeout frequency".

ACP and AC had very low timeout frequencies, allowing us to approximately peg the 'true' half solvable M at around 23. All the other combinations had greater timeout frequencies, which resulted in their reported half solvable Ms to be slightly earlier at 20 and 18 for DH and LCV, and 22 -23 for MRV, BT, and FC. For MRV, BT, and FC it appears that the majority of timeouts happened on zero solution puzzles so their "reported half solvability" came in very close to the "true half solvability".

Part 5: Monster Sudoku for N > 9

It ran reliably and correctly.			Largest Reliably Solvable N	Mean (sdev) Total Time	Mean (Sdev) Search Time	Hardest M Value (Hardest R)	Half Solvable M (Half R)	What was it?
Yes	Partly	No						
Extra Credit Monster Sudoku only								
X			8	6239 (15641)	6236 (15641)	15 (0.23)	20 (0.31)	**BT**
X			8	5729 (15023)	5727 (15023)	15 (0.23)	20 (0.31)	**BT+FC**
Extra Credit Monster Sudoku plus combinations 3-7								
X			8	9085 (18534)	9080 (18534)	15 (0.23)	20 (0.31)	**BT+FC+MRV**
X			6	744 (3521)	743 (3521)	5 (0.14)	11 (0.31)	**BT+FC+MRV+DH**
X			6	864 (4089)	862 (4089)	5 (0.14)	11 (0.31)	**BT+FC+MRV+DH+LCV**
X			10	4971 (15765)	4893 (15765)	22 (0.22)	31 (0.31)	**BT+FC+MRV+DH+LCV+ACP**
X			10	8193 (18640)	8108 (18639)	30 (0.30)	31 (0.31)	**BT+FC+MRV+DH+LCV+ACP+AC**

Answer these questions:

1. **Which more sophisticated methods let you to reach larger values of N for "hardest" puzzles?**
   ACP and AC were my best performing combinations and the only ones to surpass n=9. However they were both only able to reliably solve n=10 puzzles, at n=12 ACP reached the timeout limit with a frequency of 0.32 at the hardest M, while AC timed out with a frequency of 0.37.
2. **Which more sophisticated methods have overhead costs that outweigh their benefit Does the answer to this question change as N increases?**
   I found that DH and LCV had the worst overheads, likely due to my poor implementation of them. They were in fact so slow that they could only reliably solve n=6 puzzles as they would time out too frequently at anything higher. I did not find that the benefit/overhead ratio changed much as N increased.
3. **Do you get the same value of R for different N and different combinations?**
   I got similar values for R with some slight variation, appearing to range from R=0.22-0.30 with an outlier of R=0.14 at N=6. Interestingly I got a very consistent half solvable R of 0.31 across all the different N and different combinations tested.