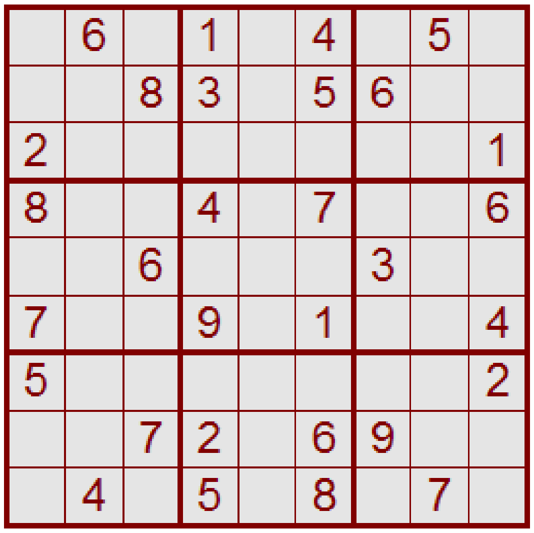
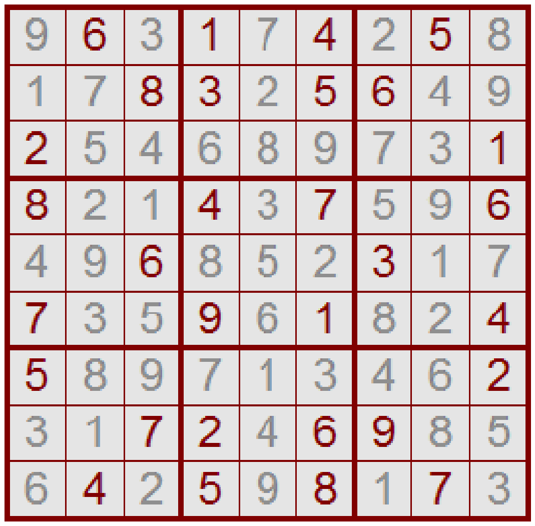
Introduction to Artificial Intelligence
Prof. Richard Lathrop
Project: Standard Sudoku Solver

1. Standard Sudoku Puzzle (Required)



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

Figure 1: Sudoku puzzle



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

Figure 2: Solution

Consider the classic 9-by-9 Sudoku puzzle in Figure 1. The goal is to fill in the empty cells such that every row, every column, and every 3-by-3 box contains exactly the digits 1 through 9, each digit occurring once. The solution to the puzzle is in Figure 2, and it satisfies these constraints:

- The digits to be entered are 1, 2, 3, 4, 5, 6, 7, 8, and 9.
- A row is 9 cells wide. A filled-in row must have exactly one of each digit. There are 9 rows in the grid, and the same constraint applies to each of them.
- A column is 9 cells tall. A filled-in column must have exactly one of each digit. There are 9 columns in the grid, and the same constraint applies to each of them.
- A box contains 9 cells in a 3-by-3 layout. A filled-in box must have exactly one of each digit. There are 9 boxes in the grid, and the same constraint applies to each of them.

You can find additional information on Sudoku at:

http://en.wikipedia.org/wiki/Sudoku
http://en.wikipedia.org/wiki/Sudoku_algorithms
http://www.sudokuweb.com/
http://4c.ucc.ie/~hsimonis/sudoku.pdf

You are to write programs to generate and solve classic 9x9 Sudoku puzzles. Note that every correct Sudoku puzzle has only **one** correct solution (see Figure 2). However, because you will generate random puzzle instances, they may have zero, one, or more solutions.

You may discuss the problem with other people, but you must develop your program and write your report independently (or with your partner, if any). You may use the Internet to learn more about Sudoku problems, and to see what solutions others have found; but you may not directly copy any code that you find online. No direct copying of code or text, ever. Both your code and your report are subject to analysis by computerized plagiarism detectors.

Extra Credit for “Monster” Sudoku
Extra credit will be given if you implement code for “Monster” Sudoku (see Extra Credit below).

Extra Credit for Local Search
Extra credit will be given if you implement code for Local Search (see Extra Credit below).

2. Constraint Satisfaction Methods (BT and FC required)

Backtracking Search (BT)
You will implement a backtracking search (BT) as described in Chapter 6 of your book.

In the absence of any other heuristics or methods (see below), this search will execute a naive “raster scan” traverse of the variables (each row left-to-right, succeeding rows top-to-bottom, and so forth). At each unassigned variable, it will iterate through the values in order 1-9 (or in odometer order if you are doing Monster Sudoku). It will assign each value in turn to that variable. If the assignment does not violate any constraints it will proceed to the next variable. If the assignment does violate a constraint it will undo that assignment and proceed to the next value. If the values for that variable become exhausted it will backtrack to the previous variable, undo its current assignment, and proceed to its next value.

In the presence of other heuristics or methods, the backtracking nature of the search will be unchanged; but the order of variables or values may change as discussed in class and your book.

Forward Checking (FC)
You will implement forward checking (FC) as described in Chapter 6 of your book.

Whenever BT assigns a value to a variable, remove any conflicting values from the domains of any other variables that participate in a constraint with the first variable. Whenever BT undoes that assignment, restore all such removed values to their respective domains (see your book and the class lecture notes).

The book-keeping necessary to achieve this correctly is a bit tricky, so plan your algorithm carefully in advance, convince yourself that it is correct, and implement it with extensive testing.

Extra Credit for Additional Heuristics or Methods
Extra credit will be given if you implement code for additional heuristics or constraint propagation methods (see Extra Credit below).

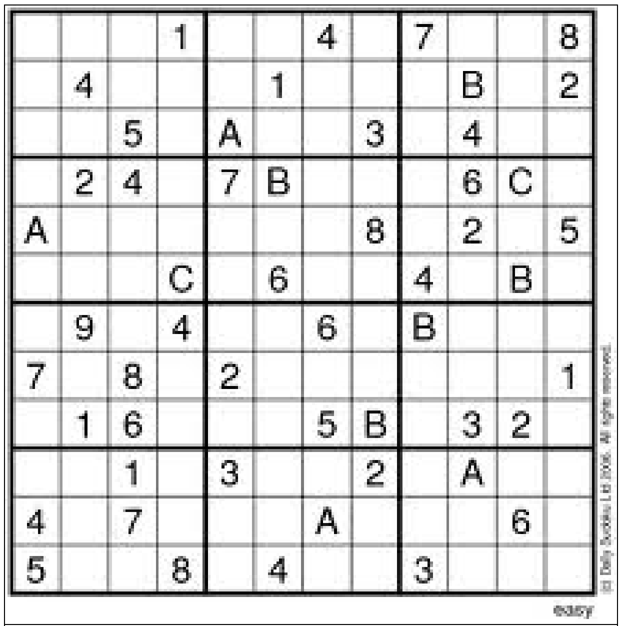
3. Extra Credit: Additional Heuristics and Methods

For extra credit you may implement additional heuristics and constraint propagation methods:

- Minimum Remaining Values (MRV) for variable ordering. The next unassigned variable chosen is one that has the fewest remaining possible values in its domain (there may be several variables that have the same minimum number of possible values remaining). Ties are broken arbitrarily, or by DH if DH also was specified.
- Degree Heuristic (DH) for variable ordering. The next unassigned variable chosen is the variable that has the highest degree to other **UNASSIGNED** variables. Note that degree is calculated only to **UNASSIGNED** variables; **assigned variables are ignored.** (If a variable has only one value remaining in its domain, then it is considered by DH to be assigned and is ignored, even if this assignment has not yet been done explicitly by BT). If both MRV and DH are specified, then MRV is done first and DH is used to break ties, with ties remaining after DH being broken arbitrarily. If only DH is specified, then only DH is used to order variables, and ties are broken arbitrarily.
- Least Constraining Value (LCV) for value ordering. The values for the chosen variable are ordered to prefer those values that delete the fewest values from the domains of all other **unassigned** variables. One easy way to implement this is to make a candidate assignment of a value to the variable, then run your Forward Checking (FC) method to eliminate conflicting values in neighboring domains, and finally count how many values remain in neighboring domains of the constraint graph — if you implement it this way, you must be sure that your book-keeping restores those deleted values after you have counted the domain sizes of the neighbors and moved on. It is OK if you implement this heuristic simply as preferring values that have the largest sum of pruned domain sizes over neighbors in the constraint graph. A better method is to prefer values that have the largest product of pruned domain sizes over neighbors in the constraint graph, because this is the size of the resulting remaining search space — if you implement it this way, you should instead sum the log of the domain sizes (because the sum of the logs is monotonic with the original product, but it avoids floating point overflow when the remaining search space is very large); and you must take care when the domain size is zero to avoid floating point errors that may occur from taking the log of zero (in this case, the resulting search space size is zero, so that candidate assignment is useless anyway). Of course, you also should check to be sure the candidate value does not already violate a constraint with other already-assigned values, because if it does it is useless.
- Arc Consistency as a Pre-processing step only (ACP). I.e., run Arc Consistency on your input puzzle only once, at the beginning before search starts.
- Arc Consistency after each step (AC). Note that if you do Arc Consistency after each step then you get the same result whether or not you do Forward Checking; but doing Forward Checking first may have efficiency advantages since it eliminates many conflicts quickly.

See Chapter 6 of your book.

4. Extra Credit: “Monster” Sudoku Puzzle



			1			4		7			8
	4				1				B		2
		5		A			3		4		
	2	4		7	B				6	C	
A							8		2		5
			C		6			4		B	
	9		4			6		B			
7		8		2							1
	1	6				5	B		3	2	
		1		3			2		A		
4		7				A				6	
5			8		4			3			

(c) Daily Sudoku Ltd 2006. All rights reserved.
easy

Figure 3: 12x12 Monster Sudoku puzzle

Monster Sudoku is a generalization of the classic 9x9 Sudoku problem that is larger and harder than classic 9x9 Sudoku. For example, consider the 12x12 Monster Sudoku puzzle shown in Figure 3. The goal is to fill in the empty cells such that every row, every column, and every 3x4 box contains exactly the digits 1 through 9 and letters A through C, each digit and letter occurring once. The solution to a “Monster” Sudoku puzzle satisfies essentially the same constraints as a standard Sudoku puzzle, generalized to an extended set of tokens that goes beyond the digits 1-9 and to boxes that may be rectangular instead of square. You can find more information below (see Specification).

You can find additional information on Monster (or Mega) Sudoku at:

http://www.dailysudoku.com/sudoku/archive.shtml?type=monster
http://www.universaluclick.com/games/sudokumonster
http://www.knightfeatures.com/KFWeb/content/features/kffeatures/puzzlesandcrosswords/KF/Sudoku/Sudoku_Monster/sudoku_monster.html
http://www.conceptispuzzles.com/index.aspx?uri=puzzle/sudoku/mega

For extra credit, you may write programs to generate and solve NxN Monster Sudoku puzzles.

5. Extra Credit: Local Search

For extra credit you may implement a local search method to solve Sudoku puzzles. You may use any of the local search methods that we covered in our unit on search. Hill-climbing with a random restart wrapper is probably a good choice.

You might create a random local search initial board position by distributing N instances of each of the first N tokens randomly across an NxN Sudoku grid. (This approach would be analogous to distributing N queens randomly across an NxN chess board in the N-queens problem.) The number of conflicts would be the number of violated constraints across all rows, columns, and blocks. (This objective function would be analogous to counting all conflicting queens.) The neighbors of a given board position would be the set of board positions that could be generated by exchanging the tokens in any pair of grid cells. (This move generator would be analogous to picking a random queen and moving her to any unoccupied square.)

A moment's thought will convince you that this approach would not be very efficient, for the same reason that the analogous approach is not very efficient for the N-queens problem. Instead, a better approach to the N-queens is to exploit the structure of the constraints. Initially, each queen is assigned to her own column and placed at a random row in her column. Then the number of conflicts need only look at horizontal and diagonal conflicts, because by construction there are no vertical conflicts. The move generator moves a queen only to another square in her own column. Can you think of an approach that exploits Sudoku constraints in analogous ways?

A better analogous approach for Sudoku exploits the constraint that each token can appear exactly once in each block (and row, and column; but for simplicity we will consider only blocks here). Initially, each set of tokens 1-N is assigned its own block and placed at random cells in that block. (Of course, if the puzzle specification placed a specific token at a specific cell, then your initial board position would put that token there and never move it afterward.) Then the number of conflicts need only look at row and column conflicts, because by construction there are no block conflicts. The move generator exchanges the tokens only in pairs of grid cells that belong to the same block. (Of course, if the puzzle specification placed a specific token at a specific cell, then your move generator would never move it afterward.)

The result of exploiting the problem constraints in this way is a more efficient problem representation.

6. Parameters

A standard Sudoku problem is defined as:

N = the number of tokens = 9.
p = the number of rows/block = 3.
q = the number of columns/block = 3.

For standard Sudoku your tokens will be the nine digits, 1-9.

To generate a random Sudoku problem you need to fill in some cells with tokens.

M = number of cells initially filled with a token.

Below, we will use zero on input to represent a blank cell.

A Monster Sudoku problem is defined by parameters N, p, and q, where N = p*q.

N = the number of tokens, and hence the edge length of the NxN grid.
p = the number of rows/block.
q = the number of columns/block.
p blocks fit across the NxN grid rows (p blocks x q columns/block = N)
q blocks fit down the NxN grid columns (q blocks x p rows/block = N)

For Monster Sudoku your first 35 tokens will be 1,2,3,...,9,A,B,...,Z.

- If you need more tokens (i.e., if N > 35), generate them odometer style:
    - 1,…,9,A,B,…,Z,11,12,…,19,1A,…,1Z,21,…,9Z,A1,…,A9,AA,…,ZZ, 111,112,…,9ZZ,A11,…,ZZZ,1111,…,ZZZZ,11111,…,ZZZZZ,…
- The Appendix contains pseudo-code to convert between odometer tokens and fixnums.

Note that you are required to use these tokens for input and output, in order to maintain consistency with published Monster Sudoku puzzles that you wish to solve. Some published Monster Sudoku puzzles also use zero as a legal token. To solve such published puzzles, you can always replace zero with the Nth token.

7. Input/Output

Your I/O will be file-based, i.e., your puzzle solver and generator will take filename arguments, and find the parameters and puzzle in those files. Use space-separated values within the files. Of course, you should check the input to verify that N = pxq, N = 9 for standard Sudoku, that M ≤ N^2, that all tokens are legal for the input value of N, and other obvious input error checking.

If an input file containing such an obvious input error is given to your program and it fails to detect the error and complain, you will lose points. Input error checking is a necessary chore for all programs. (Of course, for debugging your program during development you might temporarily wish to allow small test cases with N < 9.)

Standard Sudoku generators and solvers still must conform to the file formats given below.   For standard Sudoku, check that N = 9 and p = q = 3.

Solver

The solver will expect one filename argument. The first line of that file will be the parameters N, p, and q, separated by spaces. The N file lines that follow will give the N puzzle lines, one puzzle line on each file line. Tokens will be represented as above. Blank cells will be represented as a zero. Tokens and zeroes will be separated by spaces. For example, the input file for a 6x6 puzzle with boxes that are 2 rows by 3 columns might look like:

```
6 2 3
5 0 0 3 0 0
0 0 2 0 0 4
4 3 0 0 5 0
0 0 1 2 0 0
1 0 0 0 4 0
0 2 0 0 1 5
```

Note: one student has reported that the 6x6 puzzle above has no solution.

The solver also must provide a way to specify which heuristic constraint satisfaction methods are turned on, and which are turned off (see below). This is necessary so that you can gather timing data to analyze whether (1) any given heuristic provided a benefit in reducing overall run-time, or (2) the overhead for running that heuristic cost more time that you gained in reduced run-time.

Generator

The generator should output a puzzle in *exactly* the format that the solver will expect as input. The generator will expect two filename arguments. The first filename will be the input file with parameters N, p, q, and M, separated by spaces. For example, the input file to generate the 6x6 puzzle above, with 2x3 boxes and 14 cells randomly filled by tokens, might look like:

```
6 2 3 14
```

The second filename will be the output file to which the generated puzzle will be written. It should be written in the format that the solver will expect to read, as described above.

A generated puzzle must not violate any of the constraints as written, i.e., no row, column, or box can contain the same symbol more than once when the puzzle is generated. A randomly generated puzzle might have zero, one, or several solutions as written. In particular, it is possible (indeed, it is expected) that sometimes you will generate puzzles that do not violate any of the constraints as written — but that, after constraint propagation is run, contain one or more empty domains, and so have no solutions. The specifications are deliberately written to allow such cases, so that your constraint propagation machinery can detect such cases successfully.

Generate a random puzzle by repeatedly picking a random vacant cell and assigning a random token to it. If the token violates any of the constraints then undo that assignment, delete that token from the list of possible tokens at that cell, and repeat; if the list of possible tokens at that cell becomes empty, then fail globally (do not backtrack). Repeat until M cells have been filled. If you failed globally before the random puzzle was generated successfully, then execute a random restart and try again. Keep trying until success.

You don't really need your generator to generate puzzles with values of M that are very close to N^2. You only need it to generate values of M slightly greater than the "hardest M" value for your given N. The reason is that you only need to bracket the "hardest M" in order to conduct a local search for it within the bracket. Thereafter, you mainly will generate puzzles with M set to, or near, that "hardest M."

You easily can reason to the conclusion that generating puzzles with M = N^2 is a hard problem. To see this, notice that at some point in the generating process you would have to pass through the "hardest M" for your given N. To continue from there to M = N^2 is equivalent to solving a "hardest M" puzzle by random guessing. Note that the random generator algorithm is very simple and contains no back-tracking; so if it ever gets stuck, it stays stuck.

Consequently, be aware that if you try to generate random puzzles with M very close to N^2, it may take a very, very long time before your generator succeeds, if ever.

Compiling and Running

You must write a description of how to compile and run your programs. This description also must describe how to turn on and off the various heuristics and constraint propagation methods (see Solver above and Specification below). **Send this description to the Reader, Kartik Saxena <ksaxena@uci.edu>, no later than 27 Nov 2013, and include it in your report.**

You are allowed to use any language you wish. **However, you are responsible for ensuring that your programs compile and run on the ICS lab machines as per your description. THIS MEANS THAT YOU MUST *TEST* YOUR PROGRAMS ON THE ICS LAB MACHINES.** You will receive very little credit for your efforts if your programs cannot be compiled and run on the ICS lab machines according to your description.

Reporting

Your program must output to the screen:

- the running time in milliseconds (or fraction of)
- the number of variable/value assignments made
- whether or not a solution was found (Yes or No)
- whether or not it timed out (Yes or No)

Please use this format:

```
Time: 1002.52
Assignments: 500
Solution: Yes
Timeout: No
```

Some parts of the project report ask you to compare TOTAL_TIME and SEARCH_TIME, so for analysis purposes you must record and collect both, in one way or another (typically in a log file of some sort). Please output to the screen exactly the summary information specified by the specification in the required format.

8. Specification

Your solver must implement a backtracking search (BT) and forward checking (FC), described in your book. Your solver has a time limit, which is user-settable and defaults to one (1) minute. If your solver has not solved the puzzle within the time limit, it must time out and report failure.

You must implement various heuristics and constraint propagation methods as described below. Each heuristic and constraint propagation method must be on a switch, so that you can turn it on and off. **Include a description of how to turn them on and off in your description of how to compile and run your programs (see above).**

You will compare time/space performance with various combinations turned on and off:

1. Backtracking search (BT) only. Variables are ordered raster scan style, i.e., left-to-right and then top-to-bottom. Values are ordered odometer style, as above.

2. BT plus Forward Checking (FC) for limited constraint propagation. Whenever a value is assigned to a variable, conflicting values are removed from the domain of neighbors of that variable in the constraint graph. Whenever a value assignment is undone, those values are restored to their respective domains. Variables are ordered raster scan style, i.e., left-to-right and then top-to-bottom. Values are ordered 1-9 for standard Sudoku, or odometer style for Monster.

Extra Credit

For extra credit you may implement additional heuristics and constraint propagation methods:

3. BT+FC plus Minimum Remaining Values (MRV) for variable ordering. Values are still ordered odometer style, as above.

4. BT+FC+MRV plus Degree Heuristic (DH) to break ties for variable ordering. Values are still ordered odometer style, as above.

5. BT+FC+MRV+DH plus Least Constraining Value (LCV) for value ordering.

6. BT+ FC+MRV+DH+LCV plus Arc Consistency as a Pre-processing step only (ACP). I.e., run Arc Consistency on your input puzzle only once, at the beginning before search starts.

7. BT+ MRV+DH+LCV+ACP plus Arc Consistency after each step (AC). Note that if you do Arc Consistency after each step the result is the same whether or not you do Forward Checking; but Forward Checking may (or may not?) improve efficiency by eliminating conflicts early.

You must write your code so that you can do timing runs for the seven conditions listed above. Please use this timing rubric:

```
1.  PROGRAM START.
2.  TOTAL_START <-- current_time().
    /* current_time() returns the current time (in milliseconds or some fraction thereof) */
3.  input the problem, do all pre-processing and initialization.
4.  SEARCH_START <-- current_time().
5.  search until answer or failure.
    WHILE searching, {IF ( ( current_time() – TOTAL_START )
                                 > TIME_OUT_PARAMETER )
                              THEN time_out()
                              }.
6.  END <-- current_time().
7.  TOTAL_TIME <-- (END – TOTAL_START).
8.  SEARCH_TIME <-- (END – SEARCH_START).
9.  output the answer and record all statistics.
10. PROGRAM END.
```

Of course, during step 9 you can easily calculate other derivative statistics of interest, for example, INIT_TIME <-- (TOTAL_TIME – SEARCH_TIME), and so on. As another example, you might be curious how much of your INIT_TIME was due to input of the problem and how much was due to pre-processing and initialization, so you might set other timing variables during step 3 and record them in step 9.

Hint: JAVA is slow to allocate memory and slow to do garbage collection. Can you conserve memory, i.e., can you code it to do fewer memory allocations and fewer garbage collections? E.g., it is often better to allocate a single large static block of memory at start-up (maybe an array?) than to allocate and reclaim many small blocks of memory stepwise during the search.

9. Analysis

[Questions formerly in this section were moved to Report Template, below.]

Estimate the Critical Value of “hardest M”

Obviously, if M = 0 or M = N^2 then you can find a solution or fail very quickly. Somewhere in between those extremes there is a “hardest” value of M, i.e., a value of M for which your program takes, on average, the longest time to succeed or fail. Consider both total time and search time only. For standard Sudoku, p = q = 3. For Monster Sudoku, choose p and q to be the two factors of N closest to sqrt(N). (If you choose p =1 and q = N, you have an easy problem.)

Note that you do not need to consider values of M that are close to zero or close to N^2. You only need to bracket loosely the “hardest M,” then do local sampling within the bracketing interval to refine your estimate. For standard Sudoku, you can get an initial estimate of M by counting the number of filled cells in any published 9x9 puzzle. For Monster Sudoku, you can estimate a new value of M by M_new = M_old * ([N_new / N_old]^2). Then sample various values of M in the vicinity of your initial estimate, generating and solving 10 or more random problems for each such value of M, until you are confident you have bracketed the “hardest M.” Sample within the bracketing interval to estimate which value of M produces the longest average time. Then do additional time trials in the vicinity of that value of M in order to get a more accurate value. Use your timing data to produce the graphs required in your report below.

Estimate the Value of M for which P(solvable) = 0.5

Obviously, if M = 0 then the puzzle is solvable, so M = 0 implies P(solvable) = 1.0. As M increases, at some point P(solvable) will begin to decrease. Use the methods in the subsection above to estimate the value of M for which P(solvable) = 0.5. Is it the same as “hardest M?”

Extra Credit if You Implemented Other Heuristics or Methods

The Specification section lists five combinations of heuristics and methods beyond BT and BT+FC, which are required. You will get extra credit for those that you implement and analyze.

For example, you will want to ask whether the “hardest M” and “half-solvable M” values that you calculated above are approximately the same for every heuristic and method, or whether they vary widely. If they are about the same for every heuristic and method, then they are probably intrinsic properties of the Sudoku puzzle; while if they vary widely, then they are probably properties of the method used to solve the puzzle. For another example, you will want to know which heuristics and methods actually speed up your solver, and which have such high overhead that they actually slow down your solver.

Extra Credit if You Implemented “Monster” Sudoku

You will get extra credit if you implement and analyze “Monster” Sudoku.

For example, you will want to convert the “hardest M” and “half-solvable M” values that you calculated above into “hardest R” and “half-solvable R” values using R = M / N^2. Then you will want to ask whether “hardest R” and “half-solvable R” remains the same as N increases? For another example, you will want to ask whether the heuristics and methods with too-high overhead (i.e., those that slowed down your solver for N=9) later become useful as N increases (i.e., do they actually speed up your solver for some N>9?). For a third example, for each such combination you will want to know what is the largest N that your solver can reliably solve.

Extra Credit if You Implemented Local Search using Min-Conflicts

You will get extra credit if you implement and analyze local search using the Min-Conflicts heuristic.

You will want to ask whether your local search is faster or slower than your backtracking search for “hardest M” problems of the same size? If you also implemented “Monster” Sudoku, you will want to ask whether your local search can solve larger “hardest M” problems than can your backtracking search?

10. (Required) Report Template (What to Turn In)

[Send a copy of this page(s) to the Reader, <ksaxena@uci.edu>, by 27 November. If you change it afterward, please indicate the changed text ***clearly*** in your report (e.g., underline it and highlight the background in yellow; or use some other visually-obvious change indicator).]

[Turn in only **ONE REPORT PER TEAM** --- not per person.]
[Anything in boldface below is required; anything below in angle brackets <> is a parameter; and anything below in square brackets [] is an instruction. The safest file format for document transmission is always PDF (at least, currently). Usually, students will write the report in Word or some equivalent, and then convert the final version to PDF for submission.]

Part 1: You, and How to Run Your Code
[It is OK to use more than 1 page, if needed]

My name, ID#, UCInetID: <Mary Roe, 99999999, mroe@uci.edu>
Partner name, ID#, UCInetID (or "none"): <John Doe, 88888888, jdoe@uci.edu> or "none"
By turning in this assignment, I/We do affirm that we did not copy any text or data except CS-171 course material provided to us by the textbook, class website, or teaching staff.

The programming language(s) you used in your project:
[please, describe very clearly; write "N/A" if not applicable]
The environment needed to compile and run your project:
[please, describe very clearly; write "N/A" if not applicable]

The steps needed to compile your Generator:
[please, describe very clearly; write "N/A" if not applicable]
The steps needed to compile your Solver:
[please, describe very clearly; write "N/A" if not applicable]
The steps needed to run your Generator:
[please, describe very clearly; write "N/A" if not applicable]
The steps needed to run your Solver:
[please, describe very clearly; write "N/A" if not applicable]
If you implemented Local Search; The steps to compile/run your Local Search Solver:
[please, describe very clearly; write "N/A" if not applicable]

How to turn on and off the various heuristics and methods that you implemented:
[please, describe very clearly; write "N/A" if not applicable]

A small write up of your implementation.
[Design/Implementation/Main Classes, etc./Code Comments — Code comments should be interspersed within your code. They are not needed in your Report. Comment for ***readability***.
Folder Structure:

/SRC/ All your source code.
/BIN/ Executables.
/DOC/ Documentation (Report , HOW_TO_RUN, etc.).]

Part 2: (Required) What You or Your Team Did
[It is OK to use more than 1 page, if needed]

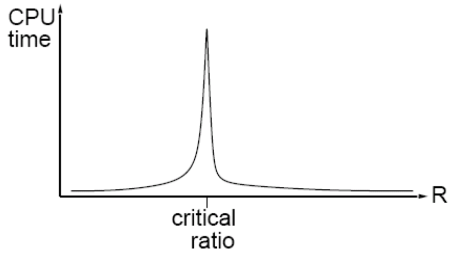
[**Please note:** You will lose many points if you claim to have done something the Reader cannot reproduce. Please check **exactly one** of Yes/Partly/No for **each** item and condition below.
Please note: If you answered "I/We tested it thoroughly" as not "Yes" then you ***must*** answer "It ran reliably and correctly" as "No" because "Not tested thoroughly" ⇒ "Not reliable."
You will lose points if you say it ran reliably and correctly but you did not test it thoroughly.]

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									
									Sudoku Puzzle Generator [15 pts]
									Sudoku Puzzle Solver [15 pts]
									Backtracking Search (BT) [15 pts]
									Forward Checking (FC) [15 pts]
Extra Credit Bonus Points									
									Minimum Remaining Values (MRV) [plus 5 pts]
									Degree Heuristic (DH) [plus 5 pts]
									Least Constraining Value (LCV) [plus 5 pts]
									Arc Consistency as a Pre-processing step only (ACP) [plus 5 pts]
									Arc Consistency after each step (AC) [plus 5 pts]
									"Monster" Sudoku Puzzle Generator [plus 5 pts]
									"Monster" Sudoku Puzzle Solver [plus 5 pts]
									Local Search using Min-Conflicts [plus 5 pts]

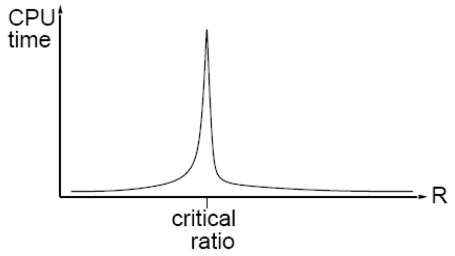
Other Extra Effort or Creativity Not Reflected Above:
[If you/your team made any extra effort, or exhibited any extra creativity, that is not reflected above, please describe it briefly but clearly here.]

Part 3: (Required) The Critical Value of “Hardest M” for N = 9 <u>[It is OK to use more than 1 page, if needed]</u>

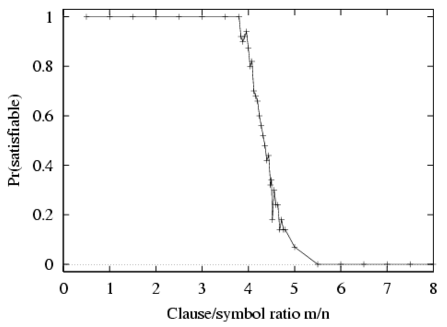
1. (Required) Find the critical value of M for N = 9 and BT (combination #1 above).
[**10 pts.** Produce a graph similar to that shown below, for total time, where R = M / N^2.]



2. (Required) Find the critical value of M for N = 9 and BT with FC (BT+FC, #2 above).
[**10 pts.** Produce a graph similar to that shown below, for total time, where R = M / N^2.]



3. (Required) How does puzzle solvability for BT+FC vary with R = M / N^2?
[**10 pts.** Produce a graph for BT+FC, similar to that shown below, where R = M / N^2.]



[For you, the x-axis will be R = M / N^2, and the y-axis will be the probability that your randomly generated standard Sudoku puzzles is solvable at all using BT+FC.]

4. (Required [10 pts, 2 pts each]) Answer these questions (for N=9):

4.a. (Required [2 pts]) Do you get the same (or approximately the same) critical value of “hardest M” for BT (in 1) as you did for BT+FC (in 2)?
4.b. (Required [2 pts]) Do you get the same (or about the same) critical value of “hardest M,” using BT+FC (in 2), for search time only? What is that critical value for search time?
4.c. (Required [2 pts]) What percent of “hardest M” puzzles for BT+FC are solvable at all?
4.d. (Required [2 pts]) For what value of M does P(solvable)=~0.5 occur?
4.e. (Required [2 pts]) 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?

Part 4: (Extra Credit) Combinations #3-7 Above for N = 9
[It is OK to use more than 1 page, if needed]

(Extra Credit, 1 pt each method) Find mean times, "hardest M," and "half-solvable M" for N = 9 and every other combination that you implemented (beyond BT+FC, i.e., #3-7 above). [Produce a table similar to that shown below. Mean Total/Search Times must be computed at the "hardest M" for that particular combination. Show the standard deviation (sdev) with each mean. Write "N/A" if not applicable. If you did not do this part, leave it blank.]

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					
							BT+FC+MRV Add Minimum Remaining Values (MRV) to BT+FC **[extra credit plus 1 pt]**
							BT+FC+MRV+DH Add Degree Heuristic (DH) to BT+FC+MRV **[extra credit plus 1 pt]**
							BT+FC+MRV+DH+LCV Add Least Constraining Value (LCV) to BT+FC+MRV+DH **[extra credit plus 1 pt]**
							BT+FC+MRV+DH+LCV+ACP Add Arc Consistency as a Pre-processing step only (ACP) to BT+FC+MRV+DH+LCV **[extra credit plus 1 pt]**
							BT+FC+MRV+DH+LCV+ACP+AC Add Arc Consistency after each step (AC) to BT+FC+MRV+DH+LCV+ACP **[extra credit plus 1 pt]**

Answer these questions:
1. Which more sophisticated methods let you solve N = 9 for "hardest M" puzzles faster?
2. Which more sophisticated methods have overhead costs that outweigh their potential benefit? (I.e., they make your solver run slower.)
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?
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?

Part 5: (Extra Credit) "Monster" Sudoku for N > 9
[It is OK to use more than 1 page, if needed]

(Extra Credit, 1 pt each method) Find mean times, "hardest R," and "half-solvable R" for N = 9 and every other combination that you implemented (beyond BT+FC, i.e., #3-7 above). [Produce a table similar to that below. Largest Reliably Solvable N and Mean Total/Search Times must be computed at the "hardest R" and largest value of N that is reliably solvable for that particular combination, where R = M / N^2, within the time limit. "Reliably Solvable" is ≥ 90% solvable. Show the standard deviation (sdev) with each mean. Write "N/A" if not applicable. If you did not do this part, leave it blank.]

It ran reliably and correctly.			**Largest Reliably Solvable N**	**Mean (Sdev) Total Time**	**Mean (Sdev) Search Time**	**Hardest M Value**	**Half-Solvable M Value**	**What was it?**
Yes	Partly	No						
Extra Credit ("Monster" Sudoku only)								
								BT [extra credit plus 1 pt]
								BT+FC [extra credit plus 1 pt]
Extra Credit ("Monster" Sudoku plus combinations #3-7 above)								
								BT+FC+MRV [extra credit plus 1 pt]
								BT+FC+MRV+DH [extra credit plus 1 pt]
								BT+FC+MRV+DH +LCV [extra credit plus 1 pt]
								BT+FC+MRV+DH +LCV+ACP [extra credit plus 1 pt]
								BT+FC+MRV+DH +LCV+ACP+AC [extra credit plus 1 pt]

Answer these questions:

1. Which more sophisticated methods let you to reach larger values of N for "hardest" puzzles?

2. Which more sophisticated methods have overhead costs that outweigh their benefit? (I.e., they make it slower.) Does the answer to this question change as N increases?

3. Do you get the same value of R for different N and different combinations?

Part 6: (Extra Credit) Local Search for Sudoku
[It is OK to use more than 1 page, if needed]

(Extra Credit, 1 pt each method) Find mean times, "hardest R," and "half-solvable R" for N = 9 and every other combination that you implemented (beyond BT+FC, i.e., #3-7 above). [Produce a table similar to that below. Largest Reliably Solvable N and Mean Total/Search Times must be computed at the "hardest R" and largest value of N that is reliably solvable for that particular combination, where R = M / N^2, within the time limit. "Reliably Solvable" is ≥ 90% solvable. Show the standard deviation (sdev) with each mean. Write "N/A" if not applicable. If you did not do this part, leave it blank.]

It ran reliably and correctly.			**Largest Reliably Solvable N**	**Mean (Sdev) Total Time**	**Mean (Sdev) Search Time**	**Hardest M Value**	**Half-Solvable M Value**	**What was it?**
Yes	Partly	No						
Extra Credit (Local Search for Standard Sudoku only)								
			9					**Local Search + Min-Conflicts for N=9 [extra credit plus 1 pt]**
Extra Credit (Local Search for "Monster" Sudoku)								
								Local Search + Min-Conflicts for N>9 [extra credit plus 1 pt]

Answer these questions:

1. Is your local search is faster or slower than your backtracking search for "hardest M" problems of the same size? Is this difference statistically significant (two-tailed test)?

2. If you also implemented "Monster" Sudoku, can your local search can solve larger "hardest M" problems than can your backtracking search?

Appendix: Convert Odometer to/from Unsigned Fixnum

Utility Functions

```
PROCEDURE ODOMETER_DIGIT_TO_FIXNUM (character ODIGIT)
/* Convert an odomoter digit, a character 1...9A...Z, to an unsigned fixnum digit, 0...34. */
RETURN POSITION( ODIGIT, "123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
/* POSITION returns the index of the first argument within the second argument. */
END

PROCEDURE FIXNUM_DIGIT_TO_ODOMETER (fixnum FDIGIT)
/* Convert an unsigned fixnum digit, 0...34, to an odomoter digit, a character 1...9A...Z. */
RETURN ELEMENT( FDIGIT, "123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
/* ELEMENT returns the character in the second argument indexed by the first argument. */
END

PROCEDURE ODOMETER_COUNT_BELOW_LENGTH (fixnum LENGTH)
/* Count the odometer numbers that can be expressed using fewer than LENGTH digits. */
BASE := 35;
POWER := 1;
ANSWER := 0;
TEMP := LENGTH - 1;
UNTIL ( TEMP <= 0 ) {
        POWER := POWER * BASE;
        ANSWER := ANSWER + POWER;
        TEMP := TEMP - 1;
        };
RETURN( ANSWER );
END

PROCEDURE ODOMETER_AS_BASE35 (string ONUMBER)
/* Treat ONUMBER as a base 35 number using ODOMETER_DIGIT_TO_FIXNUM. */
BASE := 35;
POWER := 1;
ANSWER := 0;
NDX := LENGTH( ONUMBER ) - 1;
UNTIL ( NDX < 0 ) {
        ANSWER := ANSWER
                  + ODOMETER_DIGIT_TO_FIXNUM( ELEMENT( NDX, ONUMBER ))
                    * POWER;
        POWER := POWER * BASE;
        NDX := NDX - 1;
        };
RETURN( ANSWER );
END
```

```
PROCEDURE NUMBER_OF_ODOMETER_DIGITS (fixnum FNUMBER)
/* The number of odometer digits that will be required to represent FNUMBER. */
ANSWER := 1;
UNTIL ( ODOMETER_COUNT_BELOW_LENGTH( ANSWER + 1 )
                              >= ( FNUMBER + 1) )
                    {
                    ANSWER := ANSWER + 1;
                    };
RETURN( ANSWER );
END

PROCEDURE BASE35_AS_ODOMETER (fixnum BASE35_NUMBER, fixnum LENGTH)
/* Return a string that represents BASE35_NUMBER in base 35 with odometer digits. */
BASE := 35;
ANSWER := MAKE_STRING( LENGTH );
/* MAKE_STRING returns a garbage string the length of its argument. */
POWER := EXPT( BASE, LENGTH - 1 );
/* EXPT raises its first argument to the power of its second argument. */
TEMP := BASE35_NUMBER;
DOTIMES ( NDX, LENGTH ) {
/* DOTIMES is a special form that iterates its first argument */
/* from zero upward by one for the number of times of its second argument. */
          FDIGIT := FLOOR( TEMP / POWER );
          /* FLOOR is the standard mathematical function. */
          ANSWER[ NDX ] = FIXNUM_DIGIT_TO_ODOMETER( FDIGIT );
          TEMP := TEMP - FDIGIT * POWER;
          POWER := POWER / BASE;
          };
RETURN( ANSWER );
END
```

Odometer to Fixnum

```
PROCEDURE ODOMETER_NUMBER_TO_FIXNUM (string ONUMBER)
/* Convert an odometer number, represented as a string, to an unsigned fixnum. */
COUNT_BELOW := ODOMETER_COUNT_BELOW_LENGTH( LENGTH( ONUMBER ));
ANSWER := COUNT_BELOW + ODOMETER_AS_BASE35( ONUMBER );
RETURN( ANSWER );
END
```

Fixnum to Odometer

```
PROCEDURE FIXNUM_TO_ODOMETER_NUMBER (fixnum FNUMBER)
/* Convert an unsigned fixnum to an odometer number, represented as a string. */
NDIGITS := NUMBER_OF_ODOMETER_DIGITS( FNUMBER );
COUNT_BELOW := ODOMETER_COUNT_BELOW_LENGTH( NDIGITS );
ANSWER := BASE35_AS_ODOMETER( (FNUMBER - COUNT_BELOW), NDIGITS );
RETURN( ANSWER );
END
```