Sudoku
Math 380 Final Project
Parker Joncus and Monica Yun

Introduction

Background:

Sudoku is a classic logical puzzle based on placing integer values 1-9 in a 9x9 grid. An unsolved puzzle will consist of a 9x9 grid partially filled with integers 1-9 with some spaces left blank. The objective is to fill the grid such that the numbers 1-9 appear only once in each row, column, and 3x3 sub-section of the grid.

History & Mathematical Relevance:

Sudoku puzzles became popular in the 1980's in Japan and first appeared in a U.S. newspaper in 2004. The puzzles became mainstream in the United States with efforts from Wayne Gould, the first person to write code to produce unique Sudoku puzzles rapidly.

Many mathematical ideas can be applied to the concept of Sudoku puzzles to gain a better understanding of the puzzle. For example, a few sources *(1), (5)* described a completed Sudoku board as a graph coloring problem. Each digit can be thought of as a different color and the solved game board must be filled in such that no two colors are in the same row, column or 3x3 subsection i.e. there exists an edge between two nodes if they are in the same row or column or 3x3 subsection.

Restatement of Problem:

Our goal for this project was to create an algorithm that would produce Sudoku puzzles of four levels of difficulty. This included generating solved puzzles, unsolved puzzles, creating an algorithm to solve puzzles and a system to grade the difficulty of puzzles. The puzzles were to have unique solutions and our algorithm was made as efficient as possible keeping time and space in mind.

Methods:

Our first step in this project was to construct a solved Sudoku puzzle board. Puzzle solutions have constraints that made this a more difficult task than just filling in a 9x9 matrix. From our puzzle solution, we removed symmetric elements iteratively meaning, if the element in position (3,1) was deleted, the element in position (6,9) would also be deleted. After each deletion the puzzle was tested for uniqueness. To do this we had to create an algorithm to solve any Sudoku puzzle.

We used a 'brute force' method solver we got the idea from *(6),* in this method blank spaces are initially filled in with 1. If this causes the constraints of the puzzle to be broken, the space is change to 2 this process keeps iterating until a number is filled in that doesn't conflict with the constraints of the puzzle. If none of the values 1-9 fit, the algorithm backtracks to previously filled in elements until a solution is reached. This is what we called our forward solver. To ensure the solution was unique, we created a similar solving algorithm that instead starts by filling in each element with 9 and iterating backwards. This is our backwards solver.

After an element of the puzzle was deleted, the puzzle was run through the solver to see if the puzzle still had a unique solution. If our forward solving algorithm produced the same solution as the backwards solving algorithm, the solution was unique. This is because if there was another possible solution, then forward solve and backwards solve would return different solutions since forward solve starts with 1 and iterates up and backwards solve starts with 9 and iterates down.

The last step in our process was grading the puzzles. We decided to base the difficulty of a puzzle on the Euclidean norm of the 3x3 matrix where the values in the matrix represent the number of values in the corresponding subsection. We decided to grade this way after testing the norms of unsolved puzzles of various difficulties and noticing a pattern. Easier puzzles had a higher norm and the more difficult puzzles had lower norms. Our code asks the user to select a desired difficulty. The algorithm then deletes elements until the norm is within certain bounds of the average norm for that difficulty level.

We found it could be difficult to get a puzzle with a low norm (more difficult puzzles) while maintaining a unique solution. To account for this we put a time constraint on the deleting algorithm. Once the solved puzzle enters the deleting algorithm a timer is started. If that timer exceeds a certain time while inside the deletion algorithm, it will end and generate a new puzzle. This way we avoid getting into an infinite loop or waiting too long for a result. The code ultimately returns an unsolved Sudoku puzzle of the requested difficulty.

Conclusions:

Our code produces Sudoku puzzles with four levels of difficulty, each with a unique solution. Easy and medium level puzzles are almost always produced rapidly on the first try. Hard puzzles occasionally need to be requested more than once, and expert puzzles often take a while and have to be requested more than once. This is because of grading of the puzzles based on the norm of the unsolved puzzle and the time complexity of our code. We also found that the time it took to generate a puzzle was exponentially related to the target norm of the puzzle.

If we had more time we would have done more research into different methods for solving Sudoku puzzles to possibly improve the time complexity of the code. One way we thought of to make our code faster was in the deletion algorithm. In our current algorithm when an element is deleted, we forgot to first check if that element had already been deleted. This causes our code to potentially solve the same puzzle multiple times making the run time much longer. We would insert a simple if statement to see if that node is already empty. If it is empty, there would be no need to run the solver and a new node could be selected for deletion. Also, we would do more extensive testing on our code to find the optimal time and count values to use, so that the puzzle does not take too long and it can still produce a solution. Last, we would want to explore different grading methods and see if there is a way to produce any level puzzle on the first try.

Goal

Problem Statement:

"Develop an algorithm to construct Sudoku puzzles of varying difficulty. Develop metrics to define a difficulty level. The algorithm and metrics should be extensible to a varying number of difficulty levels. You should illustrate the algorithm with at least 4 difficulty levels. Your algorithm should guarantee a unique solution. Analyze the complexity of your algorithm. Your objective should be to minimize the complexity of the algorithm and meet the above requirements. "

Analysis:

We wanted to write code to produce Sudoku puzzles with 4 levels of difficulty where each puzzle has a unique solution. In our initial research, we learned from *(4), (7)* that a Sudoku puzzle could be constructed by starting with a completely filled in Sudoku grid and systematically deleting elements until the desired level of difficulty was reached. After each deletion, the puzzle should be tested for uniqueness. If the puzzle is not unique, replace the elements that had just been deleted, then continue *(4).* This is the basic idea behind our code.

First we had to generate the solved puzzle board. We thought of a way to fill in the puzzle while keeping the constraints and then used ideas of swapping rows and columns from *(7)* to further randomize our solution.

In order to test for uniqueness, we needed a way to quickly solve puzzles. We found many techniques for solving Sudoku puzzles quickly. We based our method off the 'brute force' method from *(6).* To test for uniqueness, we implemented methods to solve the puzzle forwards and backwards and checked to see if the solutions were the same.

To come up with a grading system our first idea was from *(7),* we wanted to grade based on the techniques and logical steps required to solve the puzzle by hand. After trying to implement this we realized it would not be a realistic method of grading given the time we had to complete the project.

We then decided to test the properties of unsolved puzzles. We took 100 puzzles from *(9)* of each difficulty and created 3x3 matrices to represent the number of values in the subsection of the puzzle. We took the condition number, determinate and Euclidean norm of each puzzle. Condition number and determinate were uncorrelated to the difficulty level of the puzzle, but the norm of the puzzles were strongly correlated to difficulty. We used this as a basis for our grading method.

We learned from *(1), (4),* that Sudoku puzzles are symmetric in the empty space placement. In our code we did the same thing. When an element is deleted, so is the element that is in the mirrored position in the matrix. We deleted elements while there was a unique solution and the norm was higher than the difficulties' target. Once the norm got into the range of difficulty requested, the puzzle was be outputted.

Generating Solved Puzzle

Filling in Spaces:

After some research *(2), (3), (4), (7),* we found the first step in creating a Sudoku puzzle was to start with a solved puzzle and systematically remove elements until there are a sufficient number of empty spaces. To do this, we first had to find a way to generate a solved Sudoku puzzle.

The first challenge was to fill all 81 elements in such a way that each element in every row, column and sub 3x3 grid is unique. For our first attempt we filled in the 9x9 grid in the simplest way that would keep the constraints of the puzzle. An ordered list of integers 1-9 was used; the puzzle was filled in such a way:

Figure 1

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

Creating this kind of puzzle is not interesting since the pattern is quite obvious, it would make the resulting puzzles too easy. Our next step was to randomize the order of the digits. The algorithm is similar to the first attempt. A list of the integers 1-9 was randomly shuffled. The puzzle was filled in the same way as Figure 1, this time the numbers represent the index of the randomly shuffled list. These puzzles are still not random enough since each row will be in the same order, just with a different starting index.

Shuffling:

We wanted a way to further randomize the solved board. We learned from several sources *(5), (7),* that the rows or columns within a subsection group of three could be swapped while maintaining the constraints of the puzzle. We wrote a function (shufflePuzzle) to swap two random rows and columns in a certain sub-block. The function takes a solved puzzle board and n, the number of iterations, as inputs. Then for each sub-section of the matrix a row is chosen randomly and the other two rows are swapped. The same is done for columns. This is

done n times. In our final code we used n =100, we figured 100 iterations would be sufficient to scramble the puzzle solution.

Deletion of Nodes

Once we have a filled out puzzle, we need to delete nodes to create an unsolved puzzle. The nodes need to be deleted randomly, otherwise puzzles will be similar, and the empty spots can be uneven throughout the puzzle. There are many ways to delete the nodes randomly. We can use a random number generator to computationally "flip a coin" for each node to determine if we delete it or not. This method would not work very well in that each node would have a 50% change to be deleted. This means that if we go node by node, we would delete generally half the nodes. As we go throughout the puzzle, we would probably delete more nodes then we actually need in the puzzle and could have situations where many of the nodes on the top are deleted and none are deleted from the bottom. A way we can fix this is by changing the percentage for a node to be deleted; however, this method would take a while, and is not the easiest way to randomize the deletion.

Another easier and quicker way to delete the nodes, is to delete a node using two random integers between 1 and 9. The first random number is the row and the second random number is the column. The node in that row and column would then be deleted. This way, random positions throughout the puzzle would be deleted rather than going through each node in order.

The last method we used is an idea that we found from many websites *(2), (4), (7)*. It is the same idea as above where we use two random integer generators where the number is between 1 and 9 to generate a random row and column. The only difference is that instead of deleting just a single node, we are also deleting its mirrored pair. A node and its mirrored pair are "reflected" through the origin *(7)*. Basically, we generate a random row x and random column y, so our random node is at (x,y). The paired node is at position (10-x,10-y). This method will be faster since it deletes 2 nodes at a time. There is one exception where the very center of the puzzle (5,5) does not have a pair. In this situation, only the center node is deleted.

Puzzle Solver

Solve by Technique Method**:**

The idea behind this method was to create an algorithm that would mimic the way a human would go about solving a Sudoku puzzle. A list of problem solving techniques was found *(7)*. There is a list of techniques that can be used to solve Sudoku puzzles. Each technique has an associate cost per use. This cost is used to determine the overall difficulty of the puzzle. Techniques that are less costly are associated with easier puzzles and techniques with high cost will give the puzzle a more difficult rating.

Our idea was to write algorithms to implement each technique. Our solver would then use those techniques and grade the puzzle based on the techniques used in the solving process. The solver would first run using low cost techniques then check to see if the puzzle was still

unsolved. If the puzzle was solved it would be graded as easy. Otherwise the solver would run again using more costly techniques until the puzzle was solved.

This method of solving the puzzle appealed to us most since it would enable us to rate the puzzle's difficulty based on the logical techniques required to solve the puzzle. We wanted to grade the puzzle this way to make the difficulty based off the steps a person would make to solve the puzzle.

Attempting to create this solver, we soon realized this was not the most efficient way to solve a Sudoku puzzle with a computer. While these techniques may provide a human brain with the fastest method for solving Sudoku puzzles, implementing these techniques was slow and inefficient. We started by attempting to create an algorithm to implement the first technique, Single Candidate. About halfway through writing the code to implement Single Candidate the time complexity was already out of hand and other methods for creating solvers were pursued.

Brute Force Method**:**

Another way to solve a puzzle that is a simpler idea is the brute force method *(6)*. This method starts at the first empty node and fills the node in with the lowest value that does not cause a conflict with any other node. It will then go to the next node and fill it in with the lowest value that does not cause a conflict. This continues until either the puzzle is finished, or there is no possible value to put into a node such that there is no conflict. If there are no possible values for the current node, the method goes back to the last node that was empty and replaces the value that was originally there with the next lowest value that does not cause a conflict. If a value is found that does not have a conflict then it will continue to the next empty node. If again there is no value that does not cause a conflict, then it will go back to the last empty node before that. This process continues until the puzzle is solved. In many cases, the method will go all the way back to the first empty node and change it because it has the wrong value. This creates the circumstances where the solver gets to the very last node and then has to go all the way back to the first empty node and change it at most 8 times, since the first node can only hold 9 values. This means that depending on how many spaces are empty, the solver could take a long time if it has to go backwards far and often.

When creating an unsolved puzzle, we need to delete spaces and check to make sure that the puzzle is still unique. In order to test uniqueness of a puzzle, we created two brute force solvers. The first solver is our forward solver that was explained before. We insert the lowest value that is possible in the empty spaces and move on. The second solver is our backwards solver. We start in the same spot as the forward solver, but instead of inserting the lowest possible value in the empty spots, we insert the largest possible value for the empty slots and continue from there. If there is no possible value for the current empty position, we go back to the last empty positions and try the next largest value. The solver continues the same as the forward solver until the puzzle has a solution. If the puzzle has a unique solution, then both solvers will get the same solution. If the puzzle is not unique, then there are at least two possible solutions for the puzzle. Since one solver is forward and the other solver is backward, they will generate different solutions. The easiest way to picture this is a line where the left bound is the case where the forward solver does not have to take a step back at all. The

right bound is the case where the forward solver has to take the maximum number of steps backward, or the backward solver takes no steps backward. For a unique solution, there is one spot somewhere in the middle of the 'line' that both solutions converge to. If there are two or more solutions, one solution is closer to the forward solver and the other solution is closer to the backward solver. The solvers will stop when they reach a solution, and so they will stop at different spots on the line.

Difficulty Grading

One of the crucial questions we faced in this project was, once we have the puzzle, how we determine how difficult this puzzle is? Also, can we make a puzzle a certain difficulty if we specify what difficulty we want in the first place, or do we have to generate a puzzle, test the difficulty, and if the difficulty is not the desired level, generate a new puzzle?

We quickly had a few ideas on how to determine a puzzle's difficulty *(5), (7).* First, we thought about grading our puzzle based on how long our puzzle solver took to solve the puzzle. We also thought about grading the puzzle based on what techniques were used in the technique solver. Last, we wanted to see what relationship each level of difficulty had with the number of empty spaces in the puzzle.

Grading by Time:

Our first idea on how to grade the puzzle was by timing the solver. Our first impression was that a harder puzzle would take more time to solve than an easy puzzle. We quickly realized that the time to solve however, does not just depend on the number of nodes empty, but also on where the nodes are deleted and what values should be there. This means that grading a puzzle on the amount of time it takes for a solver to solve the puzzle would be very difficult since there are many factors that cause the time to change. An expert level puzzle could be solved just as fast as an easy puzzle if our solver does not have to backtrack. Generally, an easy puzzle should be solved faster than a harder puzzle, but the time to solve a puzzle is very inconsistent and therefore would not be a good way to grade a puzzle.

Grading by Technique:

This idea was to write an algorithm to solve the puzzle the same way a person would. We found a list of techniques used to solve puzzles *(7).* The techniques are classified by difficulty. We wanted to write an algorithm for each technique. To solve the puzzle, the algorithm would first see if the puzzle could be solved using lower level techniques. If not, the solver would be run again using more advanced techniques. Each technique would have its own difficulty grade depending on how hard the technique is. Throughout the solver these ratings would be added up and in the end we would have a puzzle difficulty level. We would then have different ranges for these ratings to determine the overall difficulty.

This idea appealed to us since it would provide a grade that was determined in the same way that a person would determine the difficulty of the puzzle. The grading would be based solely on the techniques that would be used to solve the puzzle by hand. This grading method

was abandoned after attempting to create a solver based on the techniques people use to solve puzzles by hand.

Grading by Norm:

This last method of grading we tested was in attempt to find a relationship between the number of empty spaces in the puzzle and the difficulty of the puzzle. Initially we thought more empty spaces would imply a harder puzzle. However, it does not just depend on how many spaces are empty, it also depends on which spaces are empty. Two puzzles can have the same amount of spaces empty, but have different difficulties depending on which spaces are empty. For example, consider a puzzle where the first two rows are empty and the rest of the puzzle is not. This means 18 spaces are empty. Next, consider a different puzzle where 2 spaces are empty in each subsection randomly. Now, both are mostly solved, so it seems like both should be easy, but in the first puzzle there are at least two different solutions since the two rows can be swapped after the numbers that go in the spots are determined. This means the first puzzle is impossible to solve. The second puzzle would be fairly easy to solve as long as the missing spots are truly random.

In order to test the puzzles' relationships with their difficulty, we created a 3x3 matrix for a given puzzle where each position in the matrix is the number of set numbers in the puzzle in the corresponding subsection. Figure 2 and Figure 3 show an example of the submatrix we used.

Figure 2

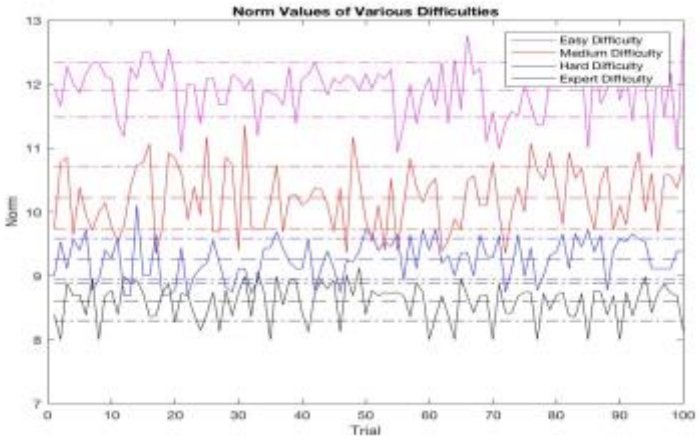
1							5	
			4		8	1	3	7
3		5	6		9	2		
					3	5		9
		1		9		4		
		2		8			6	
				3	4			8
	3		7					4
		7				6		

Figure 3

3	4	5
2	3	4
2	3	3

We took data for different properties of these matrices. These properties were the determinant, Euclidean norm, and condition number. We took data from 100 easy puzzles, 100 medium puzzles, 100 hard puzzles, and 100 extreme puzzles *(9).* From our data, we could see that the determinant and the condition number were random and did not represent the difficulty of a puzzle. However, the norm of each difficulty had a clear relationship to the difficulty of the puzzle.

Figure 4



From Figure 4, we can see that each difficulty is in its own area. There are a few areas that overlap between difficulties, but these are most likely outliers. The jagged solid lines, are the norm values all connected, piecewise. The dashed line in the middle of each area is the average norm for that difficulty. Lastly, the dot-dashed lines towards the top and bottom of each area is the range of one standard deviation in the positive and negative direction. It can be seen that the ranges for the lower and upper standard deviation bounds do not overlap, and by definition, the ranges hold about 68% of the data for its corresponding difficulty. We decided to use these standard deviation values as our grading criteria ranges. Table 1 shows the average, standard deviation and bound values.

Table 1: Norm Statistics

Difficulty	Average Norm	Standard Deviation	Upper Bound	Lower Bound	Lower Bound used
Easy	11.91	0.4285	12.34	11.48	10.71
Medium	10.22	0.4949	10.71	9.72	9.59
Hard	9.265	0.3205	9.59	8.94	8.89
Expert	8.591	0.2858	8.89	8.29	0

From Table 1, it can be seen that the bound values for each difficulty level do not overlap. Since these bound do not overlap and they are not connected, we just dropped our lower bounds for the difficulty to the upper bound of the next difficulty. This way, there are no gaps between our difficulties, and we do not have puzzles that are not graded. To do this in our program, we continued to check the norm value of the subsection matrix until the norm was less than our upper bound for that difficulty.

Figure 5

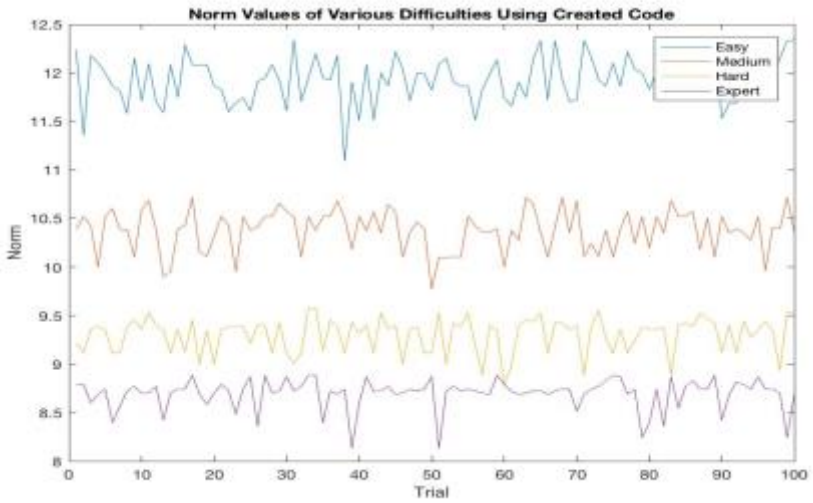


Figure 5 shows the norm values calculated using our finished puzzle generator for each difficulty. We generated 100 puzzles for each difficulty and found the norm values for each puzzle and plotted them. It can be seen that this plot is very similar to Figure 4, the norm values from an online puzzle generator *(9).* The main difference is that our plot lines are more consistent in their range and do not intersect with the other difficulty ranges very often. We also found that the mean norm of each difficulty using our program is very similar to the mean values of Figure 4. The difference in the mean can be seen in Table 2.

Table 2: Norm Statistics of Various Difficulties

Difficulty	Mean from online sources	Mean from our code	Difference	Percent Difference
Easy	11.91	11.9319	0.0219	.18%
Medium	10.22	10.3764	0.1564	1.5%
Hard	9.265	9.2999	0.0349	.37%
Expert	8.591	8.6965	0.1055	1.2%

From these results, we can conclude that our method for determining the difficulty is successful in that the mean norm for our generated puzzles is very similar to the mean norm found from online puzzles. Our norm values tend to be slightly higher than the values found from the online sources.

Norm Distribution:

The distribution of the norms of the puzzles can be roughly approximated using a Gaussian distribution. Figure 6-9 are visual representations of the norms of puzzles for each difficulty level with a normal distribution curve. We believe that as the number of puzzles used to collect data increases the distribution will converge to a Gaussian distribution.

Figure 6: Normal Distribution of Easy Puzzles

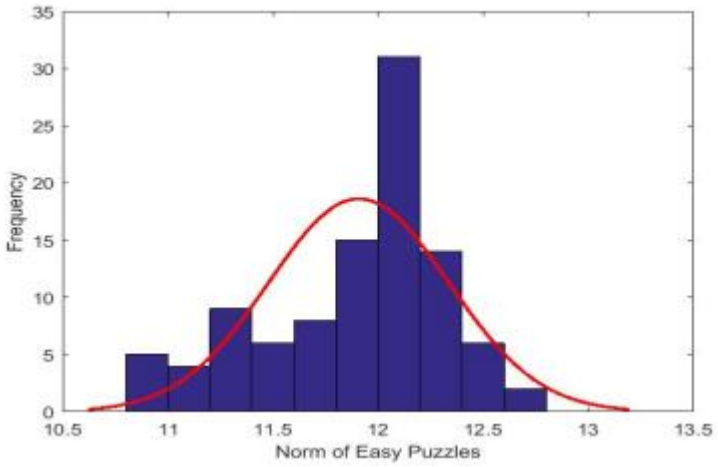


Figure 7: Normal Distribution of Medium Puzzles

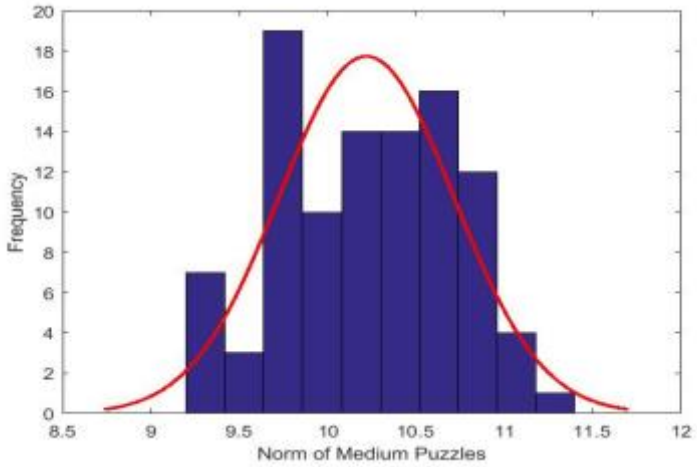


Figure 8: Normal Distribution of Hard Puzzles

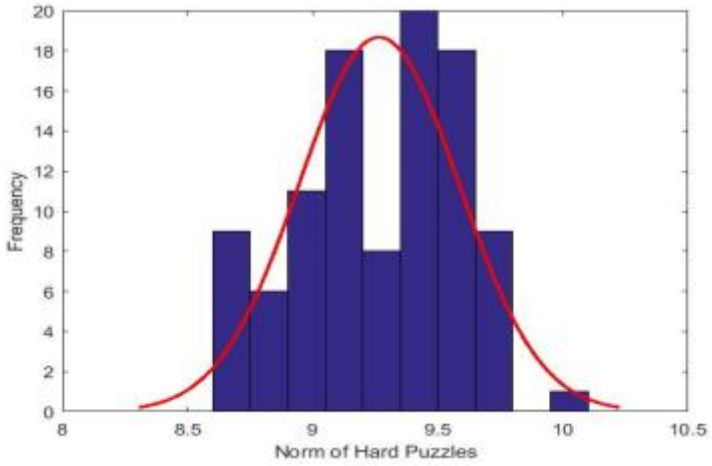


Figure 9: Normal Distribution of Expert Puzzles

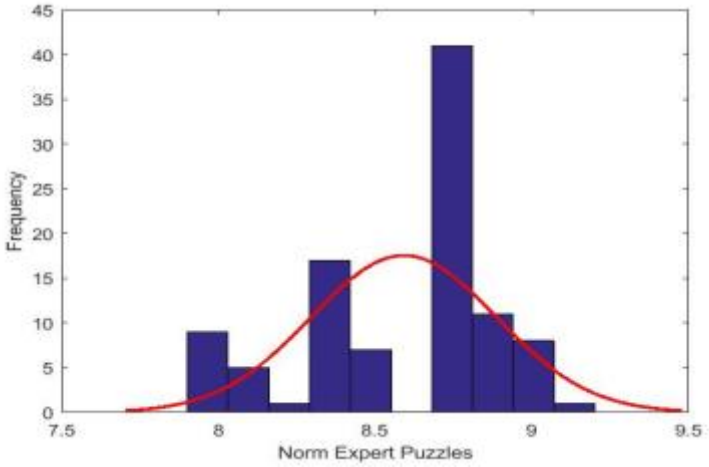
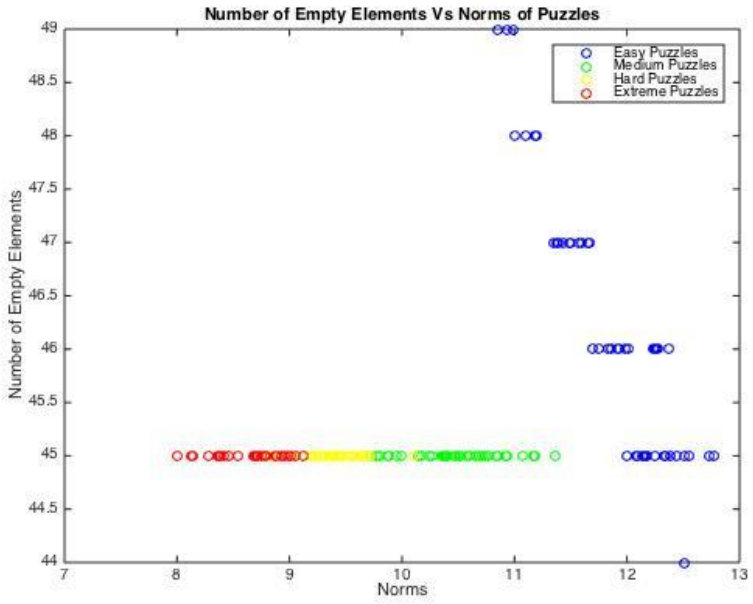


Figure 10



In Figure 10, the number of missing elements in an unsolved puzzle is compared to the norm of the matrix representing the puzzle. The minimum number of elements that could be removed was 45 for the algorithm used to create these puzzles. The most elements removed was 49, but only easy puzzles had more than 45 empty spaces. The easy puzzles with more than 45 elements removed had a smaller norm than the easy puzzles with more missing spaces. It's interesting that the norm and number of spaces missing is inversely proportional for easy puzzles. We are not completely sure why this phenomena would happen, there could have been an error as well. We would have to do further testing to determine what this means and if it is valid data.

Generating an Unsolved Puzzle

Using the ideas discussed above, we were able to create a script that generates an unsolved puzzle. The script calls upon other functions and scripts we created for each part. First, we generate a solved puzzle. After that, we prompt the user for a difficulty level and set the target value to the upper bound of that difficulties' norm. Next we randomly delete a value and its mirrored pair. We then use both our forward and backward solvers to solve the new puzzle. If the two solvers solutions are not equal, then the puzzle is not unique and we add the two deleted values back into their original spots. If the two solvers generate the same solution, then we keep the two deleted spots empty. We then update our subsection matrix and test the norm. If the norm is less than the target value, we terminate the loop and we display the puzzle. If the norm is not less than the target, it continues in the loop. The loop that we have in this script can become a problem, mostly for expert and hard difficulties. We began to wonder if it was possible for the loop to become infinite in some cases. It seems as though the answer

would be no at first, because eventually the program would delete enough spots to make the norm less than the target. However, is it possible that a puzzle with a norm higher than the target cannot have any of the values left be deleted? We were not able to determine the answer to these questions, but we assumed that the answer is yes, it is possible for the norm to never be less than the target after certain positions are deleted.

In order to prevent an infinite loop, we had two ideas. Our first idea was to have a maximum number of iterations that we are allowed to go through the loop until it is terminated. We called this value 'Count' in our code. Our other method to ensure that the puzzle being generated does not take too long, is our time constraint. For this, if the puzzle generation took too long, we would exit the loop and restart with a new puzzle. Both of these methods work, but now the question is, which method is better, and what should the parameters for the two be?

Count and Norm Constraint:

For the count method, we initialize 'Count' to 0 and after each loop, we increase 'Count' by 1. Included in the while loop is a parameter where if Count exceeds that parameter, the loop terminates. We then check to see if the norm of the matrix is less than our target, and if it is not, we restart the script and generate a new puzzle. We tested many different count parameters and found that if keeping count around 100 would work the best. During our testing we would generate an expert level puzzle, since it has the lowest target norm and the other difficulties will be more consistent, and not take as long as the hardest difficulty. We found that if we decreased the parameter to 80 or 90, it would rarely reach the target difficulty. Therefore if we cannot reach our target, the code will continue to run and will ultimately take longer. We also increased the parameter to 110 and 120. While we still would typically get a puzzle that met the target, the program would run for a much longer time, especially if we ran into one of the situations where the puzzle could not meet the target. We were not able to do extensive testing in this area due to how long the program takes to run, so all of our testing was just from running the program with different settings 5-10 times. From of this, we cannot say for certain that 100 is the optimal 'Count' parameter for us to use, but it is a fair estimate.

Time and Norm Constraint:

The time constraint is similar to the count constraint in that it limits the amount of time spent trying to create an unsolved puzzle. We do this using MATLAB's tic and toc functions. Tic starts a stopwatch, while toc tells the time on the stop watch. We use tic before the while loop and check the time at the end of each while loop, if toc is greater than our limit time, then we generate a new puzzle and restart tic. Similar to using the count, we found that if we limit the time to 10 or 15 seconds then we tend to get an unsolved puzzle and it does not take too long. If we reduce the time to 5 seconds, then we do not get solutions very often because it is not enough time. If we increase the time to 20 or 25 seconds, then it will take too long for the loop

to restart for a puzzle that cannot reach the target. We did not test to find out which time would be optimal; we just were able to observe that 10-15 seconds worked the best.

We found that using the time constraint was better than using the count as a constraint. This is because typically the reason the program takes a long time is due to the solver. Depending on which values are where, the solvers can either be fast or slow. This means that if we use count as a constraint, the program will have to use a slow solver up to 100 times. With the time constraint, the number of nodes deleted completely depends on how fast the solvers are. So if the solver is quick, a lot of nodes will be deleted and the norm will have a high chance of reaching the target in a decent time. If the solver is slow for the given puzzle, then less nodes will be deleted and will most likely not reach the target norm and restart, but since our time constraint is not very large, it will not take long to restart. Restarting gives the program a chance to generate a new puzzle that will be easier to solve. Lastly, the solver will not take a long time to run until many nodes are deleted. Obviously, if there is one empty node, it will be very fast to solve. This means that the solver will not typically take a long time until it is generally close to the target.

Another reason for the program to take a long time, is if the random row and column generator continue to generate nodes that if deleted, will cause the puzzle to be non-unique, where we have to re-insert the values and try a different random position. However, we still have to run the solvers for when these nodes are deleted, so if we continue to get puzzles that are not unique, then we are not getting closer to the target norm and using time to solve the puzzle. The time limit on the program will help limit these situations. There are incidents where we get a couple puzzles in a row that take longer than 10 seconds, and so overall they take close to a minute to generate, but generally it takes less than the 10 seconds, or somewhere between 10 and 20 seconds.

Time Analysis

Time Complexity of Generate Solved Puzzle:

Our algorithm to generate a solved puzzle has a for loop and two separate nested for loops. This means that this algorithm has a time complexity of O(n^2). In this case, our for loops are from 1-9, so our n is 9, and our algorithm is O(81).

Time Complexity of Shuffling Puzzle:

For the Shuffle function, we also have nested for loops. However, in this function, we have one for loop that goes from 1:3 and another for loop that goes from 1:m where m is the number of iterations in the function. This means we have a complexity of O($3m$), where m is an input value to determine the amount of times we need to shuffle the puzzle. In our code to generate an unsolved puzzle, we used m as 100, so the complexity would then be O(300).

Time Complexity of Solvers:

The Brute Solve method does not have any for loops, so it is a more difficult to determine the time complexity of the algorithm. Analyzing how the algorithm works, we can find the complexity of simple examples to get the complexity of the entire algorithm. If we take a puzzle that has only 1 empty spot, the worst case scenario is we try all n values, where n is 9

since there are 9 possible values. If we then have two empty spots, the worst case is that we try the first value in the first spot, and n values in the second, backtrack and try the second value in the first spot and n values in the second spot, backtrack and continue. This means we try all n values for the first spot and all n values for the second spot n times. This gives us an order of n^2. If we then have 3 empty spots, the worst case is we try all n values in the first spot, and then are left with two empty spots, which has a complexity of n^2. This means we try n^2, n times and get a complexity of n^3. Following this pattern, we can determine that the complexity of the algorithm is O(n^e) where n is the number of possible values for a single spot (n = 9) and e is the number of empty spaces. So our time complexity is O(9^e).

Time Complexity of Generating Unsolved Puzzle:

The final script uses all of the algorithms analyzed above, and it has some for loops of its own. First, it uses the function to generate a solved puzzle and the function to shuffle the puzzle. These are used separately, so their complexities are added. Next, there is a for loop and a nested for loop to generate the subset matrix. The complexity is O(n^2). After that, we have a while loop that runs x times until we get what we are looking for. Inside this while loop, we have the forward and backward solvers. We do have more for loops in the while loop, but these complexities are not as large as the complexity of the solvers, so we ignore the for loop complexity and just focus on the solver complexity. We know that the solvers both have a complexity of O(n^e). Since they are inside the while loop, their overall complexity is O(xn^e). The complexity of the entire algorithm is then 2O(n^2)+O($3m$)+O(xn^e) where n = 9, m is an input value (100 in our code), e is the number of empty spaces in the puzzle, and x is the number of times that the while loop runs. When calculating the overall complexity, we only focus on the highest order term. Since in our case m only equals 100 and n^2=81, the largest order is O(xn^e). If m was picked to be very large, then O($3m$) would be the highest order, but generally, we will not ever need to shuffle a puzzle that many times.

The complexity of this algorithm can change very easily. We know that n is a constant 9, but both x and e are variables that depend on the randomness of the numbers being deleted. This makes it very hard to predict the time to generate a puzzle because there is little consistency. Generally, easier puzzles will not need to run as long, but there are circumstance where either x or e will be high for an easier puzzle and cause the time to be large. There will also be circumstance where x or e will be small for a harder puzzle and cause the time to be short.

Time Testing and Analysis:

Figure 11

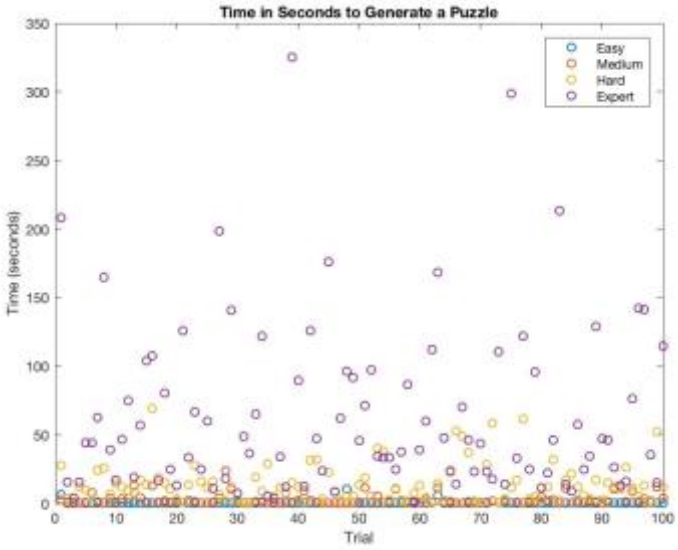


Figure 11 shows the time it takes for a puzzle to be generated using our program. We generated 100 puzzles for each difficulty and measured the time for a single puzzle to be generated. It is hard to see the easy and medium times, since the scale is larger for the expert times, but they are both along the bottom of the graph. This means that the time to generate an easy or medium puzzle is much smaller than the time to generate a hard or expert level puzzle on average. The hard and expert levels are more spread out. They both have times that are small and close to the easy and medium times, but they also have times that are really large.

Figure 12

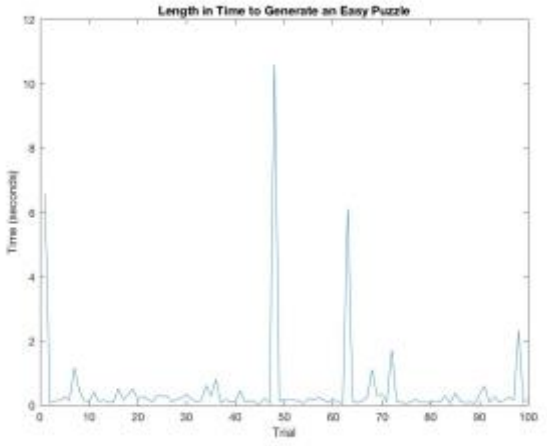


Figure 13

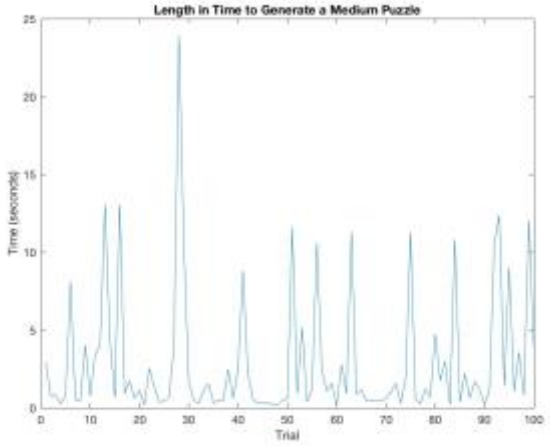


Figure 14

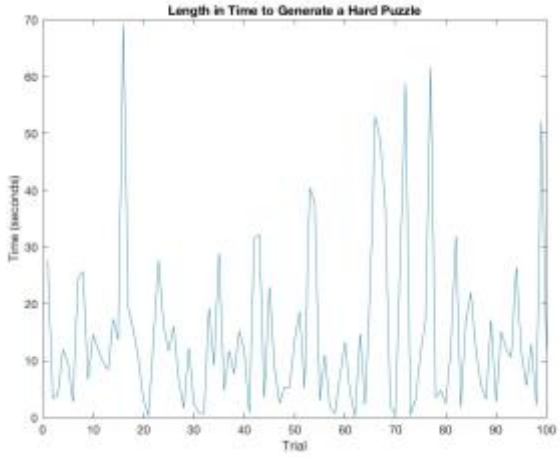
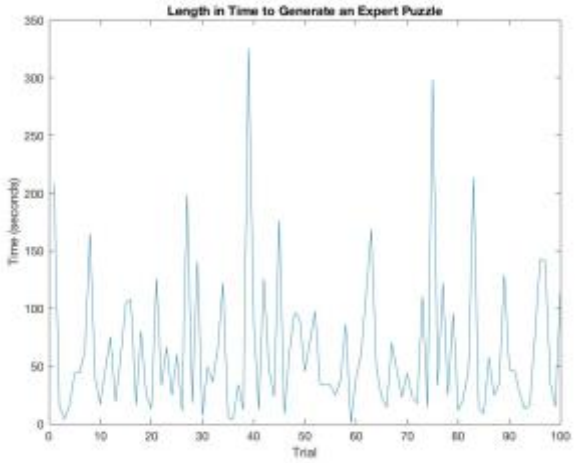


Figure 15



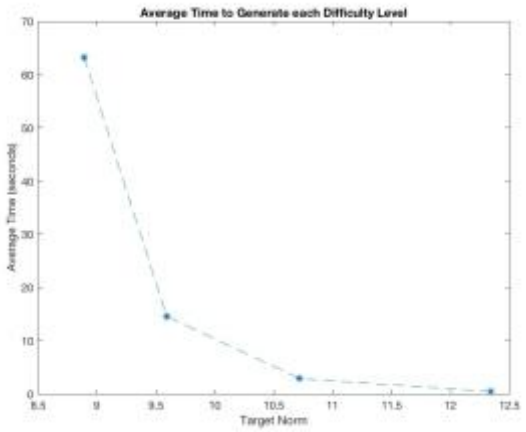
Figures 12-15 are similar to the Figure 11, except that it shows the time for each difficulty individually. These graphs help prove our reasoning for not using time to determine the difficulty of the puzzle. Table 3 shows that there are clear differences in the average time for each difficulty, but the plots show the extreme randomness and variation the time can have. All of the difficulty levels have many large spikes meaning that there is not any consistency in the time. If the times for each level did not overlap, then we could possibly grade puzzles based on the time to solve or generate. However, we have found that the times greatly overlap, and a puzzle that takes 5 seconds could be any difficulty level.

Table 3: Time Statistics of Various Difficulties

Difficulty:	Easy	Medium	Hard	Expert
Average Time:	0.4715 sec	2.8756 sec	14.5661 sec	63.2177 sec
Smallest Time:	0.05 sec	0.16 sec	0.25 sec	0.98 sec
Largest Time:	10.61 sec	23.91 sec	69.15 sec	325.15 sec

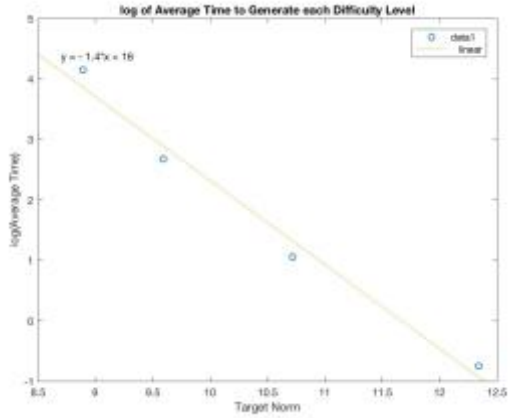
The time overlap can be seen in Table 3, where all of the difficulties had time generations lower than 1 sec. All of the difficulties had a minimum time that is less than all the average times for each difficulty, except for the expert minimum was half a second larger than the easy average. It can also be seen that Easy had a maximum time larger than the Medium average and the medium time had a maximum time larger than the Hard average, and the Hard time had a maximum larger than the Expert average. We then plotted the average times for each difficulty using the determined target norms.

Figure 16



From Figure 16, we determined that the time looks like it has an exponential decaying relationship with the norm targets used to determine the difficulty. In order to prove this relationship, we took the log of the time values and plotted it against the target norms.

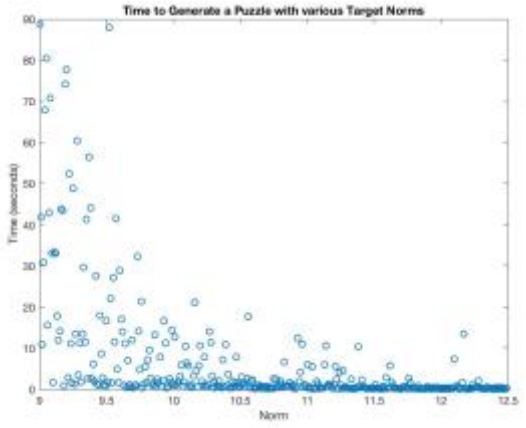
Figure 17



In Figure 17, we can see that the points are somewhat linear and the linear approximation is close to approximating each point. This means that, the average times have an exponential decaying relationship. This relationship between the log of the time and the norm can be described as the function $y = -1.4x + 16$, which is the equation of our best fit line where y is the natural log of the time and x is the norm value. Transforming the equation back into exponential form, we get the function $t = 8886110.521e^{-1.4n}$, where t is the time and n is the norm value.

From the Figures 12-15, we determined that the times to create each puzzle are very inconsistent, so this exponential decaying relationship may not hold for each individual puzzle generation. In order to see how good this relationship is with individual puzzle generations, we generated 350 puzzles and timed each one. We wanted to also test the relationship between time and different target norms, so we started with a target norm of 12.5 and after each iteration, we subtracted 0.01 from the target norm, so we end with a target of 9.0. We then plotted the time it took to generate the puzzle against the target norm.

Figure 18



Similar to before, the Figure 18 looks like it has an exponential decaying relationship, except now the points are more spread out. In order to test this relationship, we take the natural log of the time and plot it against the norm.

Figure 19

Just by looking at the data points in Figure 19, it can be seen that they do have a linear pattern, but they are very spread out. We inserted a tread line and its equation that best fit the data. The equation of the tread line is $y = -1.3x + 14$, where y is the natural log of time and x is the norm value. We then transform this function back into exponential form and get $t = 1202604.2e^{-1.3n}$, where t is the time and n is the target norm.

Table 4: Approximation Error Analysis

Figure	Equation	SSE	SST	SSR	R^2
17	$y = -1.4x + 16$	0.6410	13.3423	12.7014	0.9520
19	$y = -1.3x + 14$	547.0846	84,467	83,920	0.9935
16	$t = 8886110.521e^{-1.4n}$	801.0672	25,716	1,770.5	0.6885
18	$t = 1202604.284e^{-1.3n}$	63,052	71,138	8,086.7	0.1137

From Table 4, we can see that both the linear equations are good models for the natural log of time vs the target norm. This is because both R^2 values are close to one. We can also see that the equation $y = -1.3x + 14$ is a better approximation for this data since the R^2 value is very close to 1. We can conclude that the amount of time it takes to generate a puzzle has an exponential decaying relationship with the norm. However, the exponential equations calculated to approximate the time by transforming the linear equations, are not good approximations since their R^2 values are low. The transformation of the first equation is a decent approximation, since the R^2 value is closer to 1, but the second equation is not a good approximation of the data since its R^2 value is close to 0. The reason the transformed equation is not a good representation of our data is because when we transform the data the error increases. We do know that the data is exponentially decaying, we just cannot approximate the data very well with the transformed exponential equations. There are other equations that may fit this data more accurately, but it will be an exponential equation of the form $t = \beta e^{-\alpha n}$.

References

General Research

(1) https://en.wikipedia.org/wiki/Sudoku

(2) http://garethrees.org/2007/06/10/zendoku-generation/

Generating Puzzles

(3) http://stackoverflow.com/questions/15690254/how-to-generate-a-complete-sudoku-board-algorithm-error

(4) http://stackoverflow.com/questions/6924216/how-to-generate-sudoku-boards-with-unique-solutions

(5) http://www.math.cornell.edu/~mec/Summer2009/Mahmood/Count.html

Creating Solver

(6) https://en.wikipedia.org/wiki/Sudoku_solving_algorithmshttps://en.wikipedia.org/wiki/Sudoku_solving_algorithms

Grading

(7) https://www.sudokuoftheday.com/about/difficulty/

(8) https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

(9) http://www.websudoku.com

Computer Software

https://www.mathworks.com/help/matlab/

MATLAB 2014

MATLAB 2017

Appendix

Forward Solver (*BruteSolve*):

```
function [ solved_puzzle ] = BruteSolve( puzzle )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
empty=find(isnan(puzzle)); %%create a vector of the indicies where the empty values are
num_empty=length(empty); %%get amount of empty values
sub=zeros(3,3,9); %%initialize a list of subsets.


%%take actual subsets of the puzzle and put them in sub
for i=1:9
    for j=1:9
        sec=3*ceil(j/3)+ceil(i/3)-3;
        col=j-3*floor((j-1)/3);
        row=i-3*floor((i-1)/3);
        sub(row,col,sec)=puzzle(i,j);
    end
end


i=1; k=1; %%i=counter k=value


while sum(sum(isnan(puzzle)))>0 && i>=1 && i<=num_empty
    currentCol=ceil(empty(i)/9); %%finds current column of the empty spot
    currentRow=empty(i)-9*currentCol+9; %%find ths current row of the empty spot
    currentSec=3*ceil(currentCol/3)+ceil(currentRow/3)-3; %%finds secton number of empty spot.
    if sum(sum(sub(:,:,currentSec)==k))==1
        k=k+1;
    elseif sum(puzzle(:,currentCol)==k)==1
        k=k+1;
    elseif sum(puzzle(currentRow,:)==k)==1
        k=k+1;
    else
        puzzle(empty(i))=k;
        sub(currentRow,currentCol,currentSec)=k;
        i=i+1;
        k=1;
    end
    if k>9
        puzzle(empty(i))=nan;
        sub(currentRow,currentCol,currentSec)=nan;
        i=i-1;
        k=puzzle(empty(i))+1;
        if k>9
            currentCol=ceil(empty(i)/9);
            currentRow=empty(i)-9*currentCol+9;
            currentSec=3*ceil(currentCol/3)+ceil(currentRow/3)-3;
            sub(currentRow,currentCol,currentSec)=nan;
            puzzle(empty(i))=nan;
            i=i-1;
            k=puzzle(empty(i))+1;
            if k>9
                puzzle(empty(i))=nan;
                currentCol=ceil(empty(i)/9);
                currentRow=empty(i)-9*currentCol+9;
                currentSec=3*ceil(currentCol/3)+ceil(currentRow/3)-3;
```

```
            sub(currentRow,currentCol,currentSec)=nan;
            i=i-1;
            k=puzzle(empty(i))+1;
        end
    end
end
end

solved_puzzle=puzzle;
end
```

Backward Solver (*BruteSolveBack*):

```
function[solved_puzzle]=BruteSolveBack(puzzle)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
empty=find(isnan(puzzle));  %%create a vector of the indicies where the empty values are
num_empty=length(empty);  %%get amount of empty values
sub=zeros(3,3,9);  %%initialize a list of subsets.


%%take actual subsets of the puzzle and put them in sub
for i=1:9
    for j=1:9
        sec=3*ceil(j/3)+ceil(i/3)-3;
        col=j-3*floor((j-1)/3);
        row=i-3*floor((i-1)/3);
        sub(row,col,sec)=puzzle(i,j);
    end
end


i=1;  k=9;  %%i=counter k=value


while sum(sum(isnan(puzzle)))>0 && i>=1 && i<=num_empty
    currentCol=ceil(empty(i)/9);  %%finds current column of the empty spot
    currentRow=empty(i)-9*currentCol+9;  %%finths current row of the empty spot
    currentSec=3*ceil(currentCol/3)+ceil(currentRow/3)-3;  %%finds secton number of empty spot.
    if sum(sum(sub(:,:,currentSec)==k))==1
        k=k-1;
    elseif sum(puzzle(:,currentCol)==k)==1
        k=k-1;
    elseif sum(puzzle(currentRow,:)==k)==1
        k=k-1;
    else
        puzzle(empty(i))=k;
        sub(currentRow,currentCol,currentSec)=k;
        i=i+1;
        k=9;
    end
    if k<1
        puzzle(empty(i))=nan;
        sub(currentRow,currentCol,currentSec)=nan;
        i=i-1;
        k=puzzle(empty(i))-1;
        if k<1
            currentCol=ceil(empty(i)/9);
            currentRow=empty(i)-9*currentCol+9;
            currentSec=3*ceil(currentCol/3)+ceil(currentRow/3)-3;
            sub(currentRow,currentCol,currentSec)=nan;
            puzzle(empty(i))=nan;
            i=i-1;
            k=puzzle(empty(i))-1;
            if k<1
                puzzle(empty(i))=nan;
                currentCol=ceil(empty(i)/9);
                currentRow=empty(i)-9*currentCol+9;
                currentSec=3*ceil(currentCol/3)+ceil(currentRow/3)-3;
                sub(currentRow,currentCol,currentSec)=nan;
                i=i-1;
                k=puzzle(empty(i))-1;
            end
        end
```

```
    end
end


solved_puzzle=puzzle;
end
```

Generating Puzzle with Random Indices (*generateRandPatterned2*):

```
function puzzle = generateRandPatterned2( n )
%outputs an ordered completed sudoku puzzle


%initialize variables k, n counters and empty a 9x9 matrix of non numerical
%values
n = 1:9;
k = 1:9;
n = n(randperm(length(n)));
empty = nan(9,9);
start = 4;



for i = 1:9  %for each row, do this:
   for j = 1:9 %fill in each element in the row


         empty(i,j) = n(j);


    end
 if i==3 | i==6
    start = 5;
  else
    start = 4;
  end
   for ii = 1:9 %make list with new starting index
      rowIndex = start+ii-1;  %place of first element in new list
     if rowIndex>9
        rowIndex = mod(rowIndex,9);
      end
     k(ii) = n(rowIndex);
   end
     n = k;


end
puzzle = empty;


end
```

Shuffling Puzzle (*shufflePuzzle*):

```
function randPuzzle = shufflePuzzle( puzzle,n )
%this function takes a puzzle generated according to our pattern and
%returns a random solution board. Shuffles the board n times
for iterations = 1:n
   for i = 1:3 %three 3x3 submatrices. when rows/columns are shuffles in these submatrices the constraints of the puzzle will remain
satisfied.
       x = 3*rand;  %determines what row to keep in place (swap other two)
       y = 3*rand;  %determines what column to keep in place
       x = ceil(x);
       y = ceil(y);
       temp = nan(1,9);
       row1 = mod((x+1),3)+1+(i-1)*3;  %gets the indices of rows being swapped
       row2 = mod((x+3),3)+1+(i-1)*3;
       temprow = puzzle(row1,:);  %stores that row in temp
        puzzle(row1,:) = puzzle(row2,:);  %swap rows
        puzzle(row2,:) = temprow;
        %get indeces of columns to be swapped
       column1 = mod((x+1),3)+1+(i-1)*3;
       column2 = mod((x+3),3)+1+(i-1)*3;
       tempcolumn = puzzle(:,column1);
       puzzle(:,column1) = puzzle(:,column2);
```

```
        puzzle(:,column2) = tempcolumn;
    end
end
randPuzzle = puzzle;
end
```

Generating Unsolved Puzzle Using Time and Norm (*UnsolvedPuzzleGen2*):

```
clear;
PuzzleSolution=generateRandPatterned2; %%creates the solved puzzle using our generator
PuzzleSolution=shufflePuzzle(PuzzleSolution,100); %shuffles the puzzle
Puzzle=PuzzleSolution; %%Creates a varibale Puzzle that we can delete items from while keeping the solved puzzle
sub_matrix=9*ones(3,3); %%initializes the subset matrix used to calculate the norm to a 3x3 matix of 9's since each subsection is full
sub=zeros(3,3,9); %%initialize a list of subsets.


%%take actual subsets of the puzzle and put them in sub
for i=1:9
    for j=1:9
        sec=3*ceil(j/3)+ceil(i/3)-3;
        col=j-3*floor((j-1)/3);
        row=i-3*floor((i-1)/3);
        sub(row,col,sec)=Puzzle(i,j);
    end
end


%%Prompts user for a difficulty level
%%the target value is what the norm has to be less than for that difficulty
Difficulty=input('Enter desired difficulty (Easy=1, Medium=2, Hard=3, Expert=4): ');
if Difficulty==1
    target=12.34;
elseif Difficulty==2
    target=10.72;
elseif Difficulty==3
    target=9.59;
elseif Difficulty==4
    target= 8.89;
else
    sprintf('The value entered for difficulty is invalid.')
    target=50;  %makes sure target norm is too high so while loop does not start
end


%%Go through a loop deleting random slots and its mirrored pair, until the
%%difficulty is the right level, or count gets too high.
tic;
while norm(sub_matrix)>target
    r=randi([1,9],1,2); %creates a vector of size 2 of two random integers between 1 and 9. The first spot is the row, and the second is the
column
    r1=10-r; %%creates the mirrored verson of the generated vector r
    value1=Puzzle(r(1),r(2)); %saves the value in the original spot
    value2=Puzzle(r1(1),r1(2)); %saves the value in the mirror spot
    Puzzle(r(1),r(2))=nan; %Sets the spot in the puzzle to empty
    Puzzle(r1(1),r1(2))=nan; %Sets the mirror spot in te puzzle to empty
    ForSol=BruteSolve(Puzzle); %Solves the puzzle forward with solver
    BackSol=BruteSolveBack(Puzzle); %Solves the puzzle backward with solver
    if sum(sum((ForSol==BackSol)))~=81 %tests to see is solutions are the same
        Puzzle(r(1),r(2))=value1; %if not the same, sets the last empty spots back to their original values
        Puzzle(r1(1),r1(2))=value2;
    end


    %generates the new subset sections
    for i=1:9
        for j=1:9
            sec=3*ceil(j/3)+ceil(i/3)-3;
            col=j-3*floor((j-1)/3);
            row=i-3*floor((i-1)/3);
            sub(row,col,sec)=Puzzle(i,j);
        end
    end


    %generates the subset matix for calculating the norm
```

```
    for i=1:9
        sub_matrix(i)=sum(sum(sub(:,:,i)>0));
    end


    %%Checks to see if the puzzle is taking to long restarts if it is.
    if toc>10
        PuzzleSolution=generateRandPatterned2;  %%creates the solved puzzle using our generator
        PuzzleSolution=shufflePuzzle(PuzzleSolution,100);  %shuffles the puzzle
        Puzzle=PuzzleSolution;  %%Creates a varibale Puzzle that we can delete items from while keeping the solved puzzle
        sub_matrix=9*ones(3,3);  %%initializes the subset matrix used to calculate the norm to a 3x3 matrix of 9s since each subsection is full
        sub=zeros(3,3,9);  %%initialize a list of subsets.


        %%take actual subsets of the puzzle and put them in sub
        for i=1:9
            for j=1:9
                sec=3*ceil(j/3)+ceil(i/3)-3;
                col=j-3*floor((j-1)/3);
                row=i-3*floor((i-1)/3);
                sub(row,col,sec)=Puzzle(i,j);
            end
        end
        tic;  %restarts stop watch
    end
end


%display puzzle
Puzzle
```

Generating Full Simple Puzzle (indirect code) (*generatePatterned*):

```
function puzzle = generatePatterned( n )
%outputs an ordered completed sudoku puzzle


%initialize variables k, n counters and empty a 9x9 matrix of non numerical
%values
k = 1;
n = 1;
%n = n(randperm(length(n))
empty = nan(9,9);



for i = 1:9  %for each row, do this:
    k = n;      %this is the first element in the row
    for j = 1:9 %fill in each element in the row
        if k<=9 %we only want values 1-9
            empty(i,j) = k;
            k = k+1;
        else
            k = 1;
            empty(i,j) = k;
            k = k+1;
        end


    end
    n = k+3;  %the starting value of each row increases by 3 each time.
    if k>9
        n = 4;
    end
    if n>9
        n = mod(n,9)+1;  %want values 1-9
    end
end
puzzle = empty;


end
```

Generating Unsolved Puzzle Using Count and Norm (indirect code) (*UnsolvedPuzzleGen*):

```
clear;
PuzzleSolution=generateRandPatterned2;  %%creates the solved puzzle using our generator
PuzzleSolution=shufflePuzzle(PuzzleSolution,100);  %shuffles the puzzle
Puzzle=PuzzleSolution;  %%Creates a varibale Puzzle that we can delete items from while keeping the solved puzzle
Count=1;  %%initializes the count to 1
sub_matrix=9*ones(3,3);  %%initializes the subset matrix used to calculate the norm to a 3x3 matix of 9's since each subsection is full
sub=zeros(3,3,9);  %%initialize a list of subsets.


%%take actual subsets of the puzzle and put them in sub
for i=1:9
    for j=1:9
        sec=3*ceil(j/3)+ceil(i/3)-3;
        col=j-3*floor((j-1)/3);
        row=i-3*floor((i-1)/3);
        sub(row,col,sec)=Puzzle(i,j);
    end
end


%%Prompts user for a difficulty level
%%the target value is what the norm has to be less than for that difficulty
Difficulty=input('Enter desired difficulty (Easy=1, Medium=2, Hard=3, Extreme=4): ');
if Difficulty==1
    target=12.34;
elseif Difficulty==2
    target=10.72;
elseif Difficulty==3
    target=9.59;
elseif Difficulty==4
    target=8.89;
else
    sprintf('The value entered for difficulty is invalid.')
    target=50;  %makes sure target norm is higher than possible norm
end


%%Go through a loop deleting random slots and its mirrored pair, until the
%%difficulty is the right level, or count gets too high.
while Count<110 && norm(sub_matrix)>target
    r=randi([1,9],1,2);  %creates a vector of size 2 of two random integers between 1 and 9. The first spot is the row, and the second is the
column
    r1=10-r;  %%creates the mirrored verson of the generated vector r
    value1=Puzzle(r(1),r(2));  %saves the value in the original spot
    value2=Puzzle(r1(1),r1(2));  %saves the value in the mirror spot
    Puzzle(r(1),r(2))=nan;  %Sets the spot in the puzzle to empty
    Puzzle(r1(1),r1(2))=nan;  %Sets the mirror spot in te puzzle to empty
    ForSol=BruteSolve(Puzzle);  %Solves the puzzle forward with solver
    BackSol=BruteSolveBack(Puzzle);  %Solves the puzzle backward with solver
    if sum(sum((ForSol==BackSol)))~=81  %tests to see is solutions are the same
        Puzzle(r(1),r(2))=value1;  %if not the same, sets the last empty spots back to their original values
        Puzzle(r1(1),r1(2))=value2;
    end


    %generates the new subset sections
    for i=1:9
        for j=1:9
            sec=3*ceil(j/3)+ceil(i/3)-3;
            col=j-3*floor((j-1)/3);
            row=i-3*floor((i-1)/3);
            sub(row,col,sec)=Puzzle(i,j);
        end
    end


    %generates the subset matix for calculating the norm
    for i=1:9
        sub_matrix(i)=sum(sum(sub(:,:,i)>0));
    end


    Count=Count+1;  %increases count by 1
end
if norm(sub_matrix)>target
    sprintf('Sorry could not get desired difficulty, try again.')
```

```
    UnsolvedPuzzleGen;
end
Puzzle
```