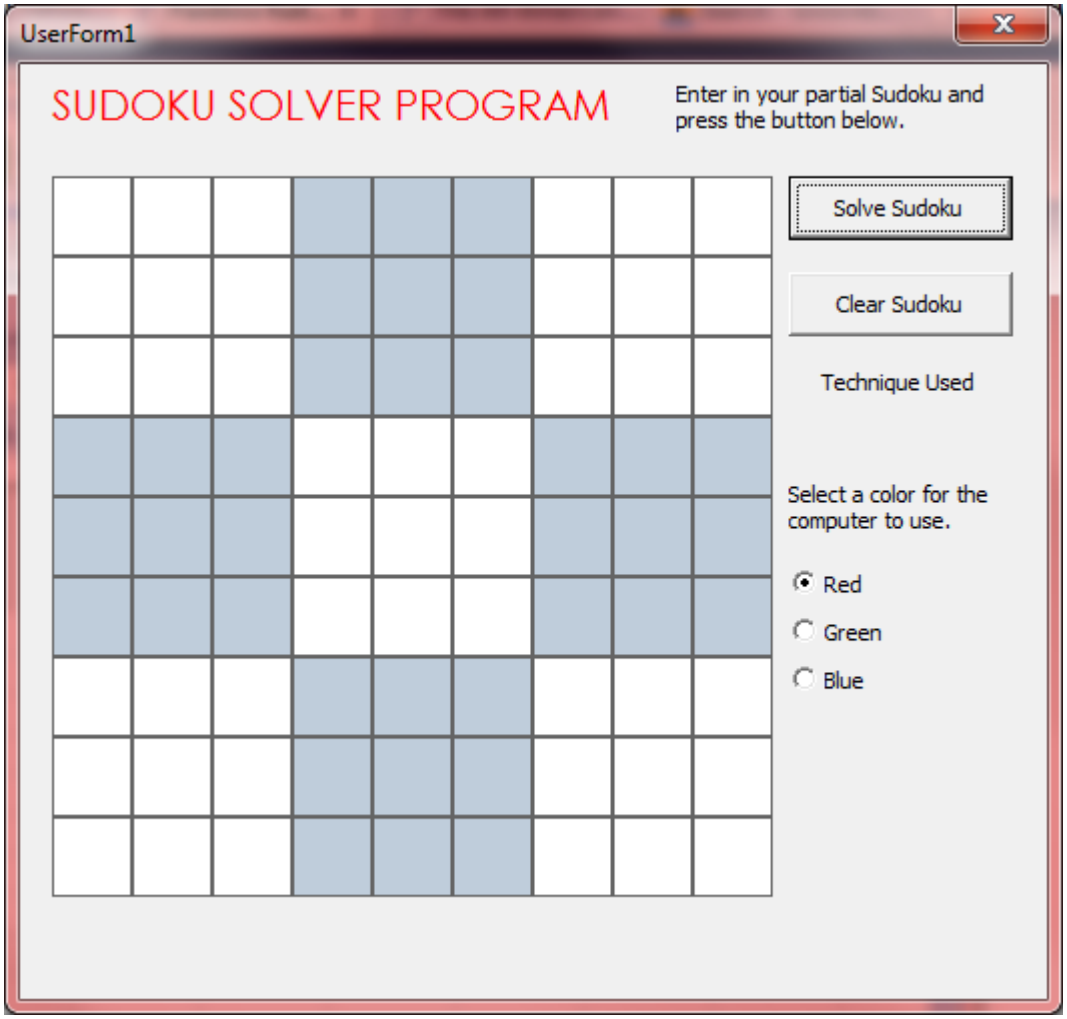
# Implementation Documentation




## Sudoku Solver Logic

Sudoku Solver uses 4 different algorithms, or “techniques” to find numbers in the Sudoku puzzle.

When the “Solve Sudoku” button is clicked, 10 arrays are made. The first array is a 2 dimensional array of the entire puzzle. The other 9 arrays represent each other “mini-squares” seen in the puzzle above.

Once the data has been collected, the program will then launch into each technique until it succeeds in finding a number. Once a number has been found the program exits and waits again for the user to click the button.

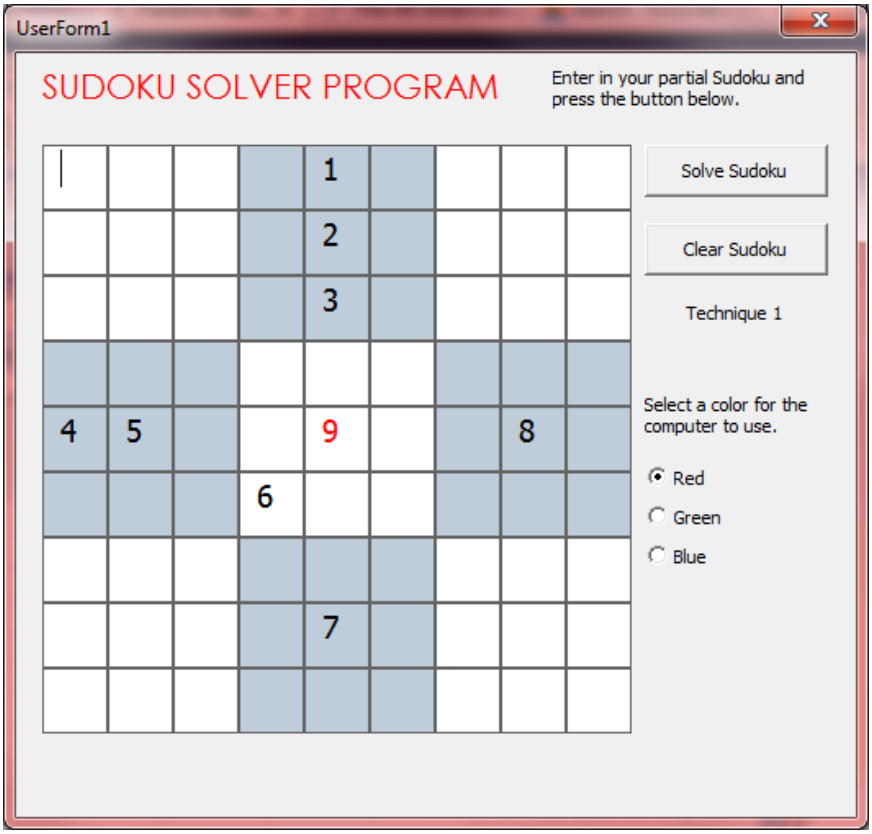
Here is a description and demonstration of how each technique works.

Technique #1

This is the simplest technique used, and also the one used most often. This technique alone is usually capable of solving easy Sudokus. It is affectionately referred to as the "brute-force" attack for solving the Sudoku.

### How it works:

This program simply cycles through each and every individual spot in the entire puzzle. If a spot is has a number already, it moves onto the next spot. Once the program reaches a blank cell, it creates an array of all the numbers that affect this particular cell, using the cell's row, column and "mini-square." Armed with that array, the program then checks to see if 8 of the 9 possible numbers are held within the array. If they are, then the 9th number must be the correct answer, and the number is entered in, and the program exits. If the program fails to find 8 of the 9 numbers, it moves onto the next cell.




This picture demonstrates this method in action.

When the program analyzed the square in the center, it discovered the 1,2,3,7 in the column. It then saw the 4,5,8 in the row and the 6 in the "mini-square".

It combined them all and recognized that it the only number missing was the "9", and inserted it into the correct spot.

Technique #2

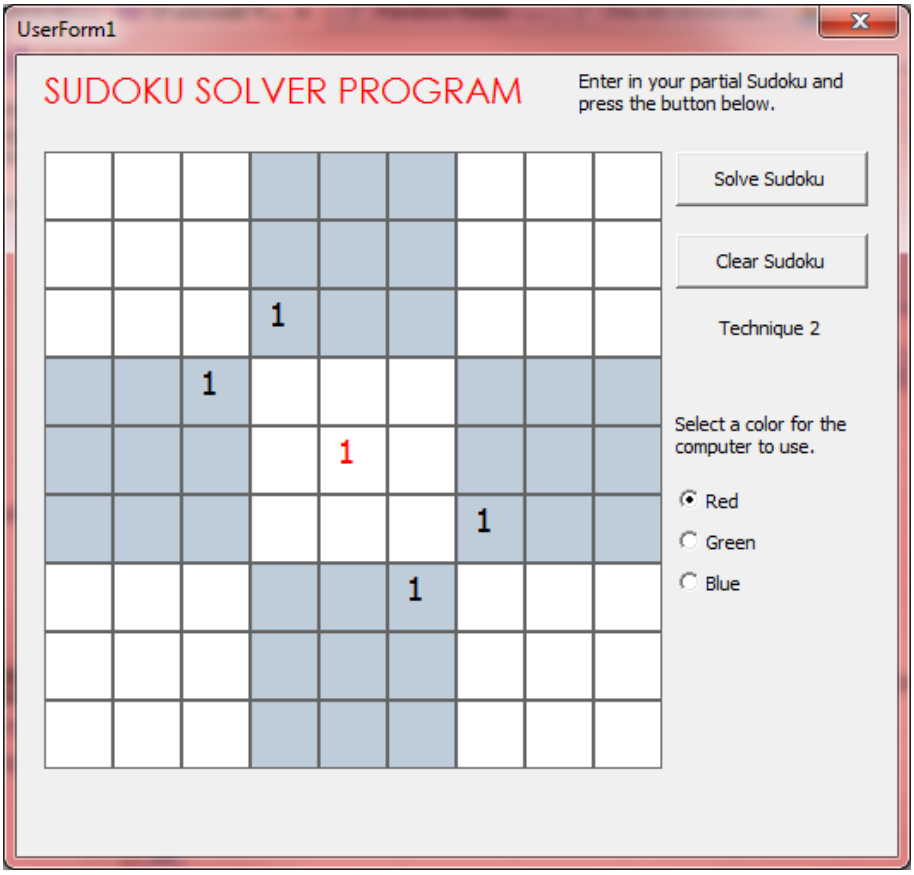
While technique 1 is adequate for finishing partially completed Sudoku's often, even on easy Sudokus, it usually hits a wall when it finds areas of puzzles that are completely empty. Technique 2 is meant to fill in extra numbers, based on columns, and hopefully make it possible to solve additional portions of the puzzle with the first technique.

### How it works:

This technique cycles through each of the columns in the puzzle. For each column, the program creates an array of the numbers in that column, the numbers needed to complete the array and the locations of the empty spots in the column.

Then, for each number missing in the column, the program also creates an array of Booleans values, used to indicate if a spots in the column are viable or not. Then for each spot in the column, the program checks against the corresponding row and "mini-square" to see if current missing number can be put there. As it does this, it tallies up how many spots are invalid. If that count reaches 8, we can know that there must be one spot that works. The program may then put the missing number in that spot and exit.

This technique, in combination with the first technique, is capable of solving a large number of puzzles.




This picture demonstrates this process.

The program cycled through columns until it got to the center column. Analyzing that column, it saw that the top 3 and bottom 3 values could not hold a "1" because there was a "1" in that "mini-square." It also saw that the space directly above and below the "1" could not work, because of 1's in the rows being checked. Thus, only one spot was possible, in the center.

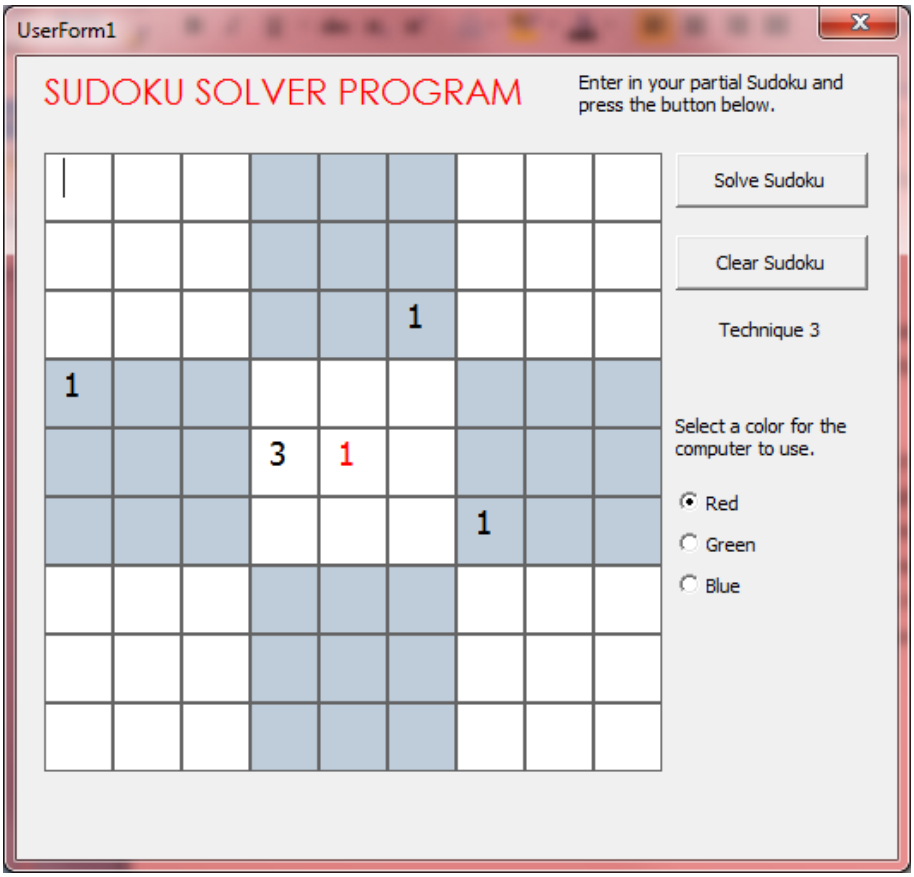
Technique #3

Having completed technique 2, it is relatively simple to add a similar technique to add an additional dimension of solving capability to the program.

**How it works:**

This technique is almost identical to technique 2, except that instead of looking at the columns in the Sudoku, it analyzes row by row. It does this in almost exactly the same fashion the technique 2 analyzes column by column.

As this method cycles through all the rows in the puzzle, it creates arrays of all the numbers seen had in the row, all the numbers missing and the indices of the empty spots in the row. Then it just runs through each possible number for that particular row. For each number it creates an array of possible places it can be in that row. For each spot, it checks its corresponding "mini-square" and column, searching for the missing number. If it finds the number, it marks the spot as false, as in, this spot cannot hold the missing number. If it finds 8 invalid locations, then the 9th must be correct.




This picture demonstrates this technique.

The computer cycled through each row until it got to the middle row. It knew that the first and last 3 values couldn't hold a 1, because for each spot in row in those "mini-squares" it identified a 1. In addition to that, a place occupied by a number, the "3" can't hold a new number. That leaves just 2 possible spots, one of which is invalidated by a 1 in the column—leaving the spot where just the "1" currently is.

Technique #4

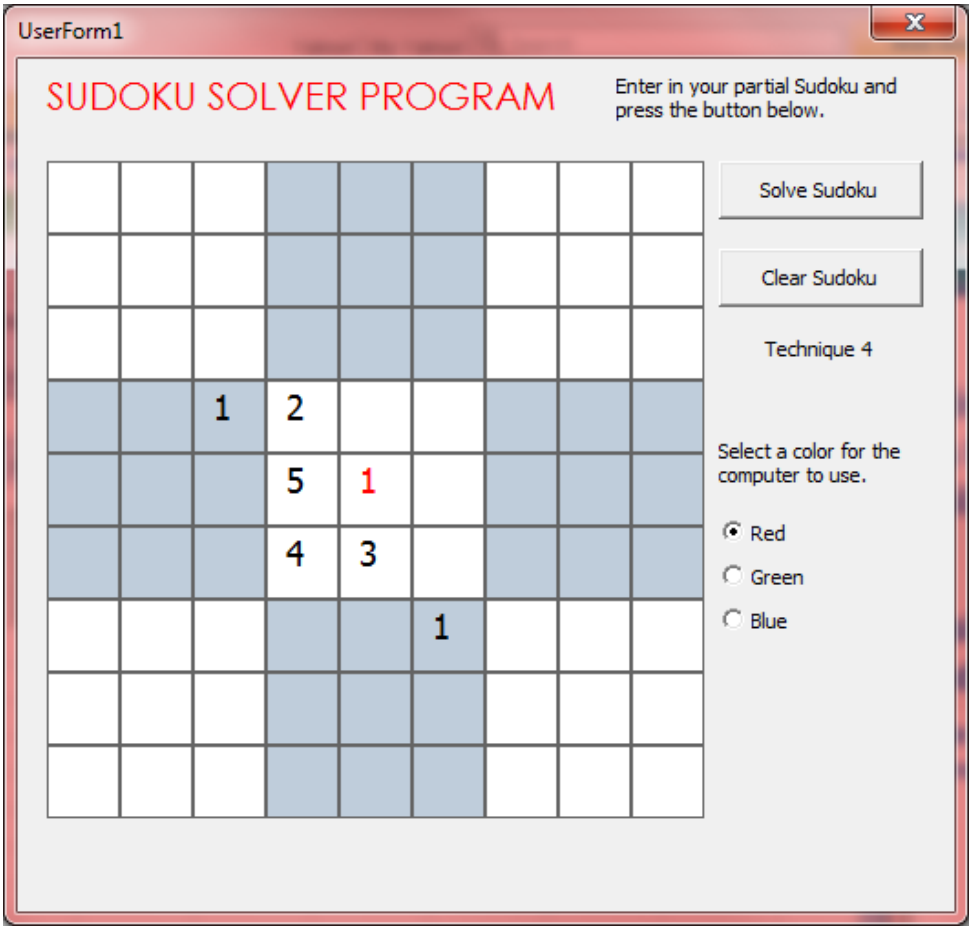
With just the first three solving solutions, almost all but the most difficult Sudoku's can be solved. While this last technique used in the program is not the catch-all for Sudokus, it does give the program one more additional algorithm it can run in the event the first three fail to find possible numbers.

### How it works:

This technique relies on the same principles used in techniques 2 and 3, but instead of looking row by row, or column by column, it looks at each "mini-square" one by one for possible solutions.

It does all of the exact same things as mentioned in techniques 2 and 3, in terms of how it identifies correct number locations, but the process is complicated slightly by the fact that a "mini-square" is a 2 dimensional array, and the above processes dealt strictly with 1 dimensional arrays.

To make this function work, I first had to know which "mini-square" was being analyzed, to correctly select the row and column that should be used in validating each spot in the "mini-square." In addition to that, some conversion, from one dimension to another, was required, both in the inputs and outputs.




This picture demonstrates how this technique works.

In looking at the center square, the program looked at each spot, and knew the ones with numbers were invalid. In receiving this particular square, it offset the rows and columns by 3, thus becoming aware of the 1's that invalidate every spot in the "mini-square", except for where the "1" currently is.

## Difficulties Encountered

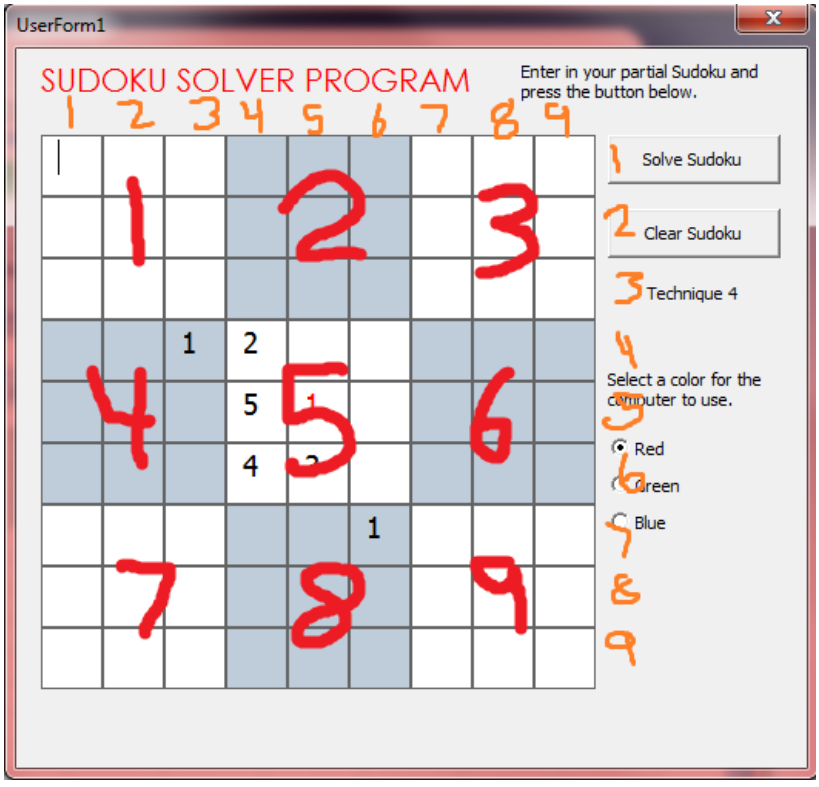
When I first launched into this project, I did so attempting to implement Technique 2, as described above—initially failing quite miserably, because I did not have a well-established conceptual plan. However, learning from my difficulties, I had the idea to create Technique 1, which I was able to do relatively quickly and easily, from my initial experience of mucking around in a bad plan.

I quickly saw that, while it was okay, it still had some severe deficiencies, so I returned to my previous attempts at solving by column. This was probably the most difficult part to accomplish for this project, simply because of all the nested loops, If Statements and complex arrays—all interacting with each other. It was not until I finally gave up and drew a simple diagram, mapping out the entire process, that I was able to successfully complete this stage of the program.

With Technique 2 completed, it was relatively simple to convert it to work with rows, instead of columns.

The last major conceptual hurdle took place when I tried to adapt the above process to work with "mini-squares" rather than rows or columns. This was mainly difficult for 2 reasons.

1. I had all the "mini-squares" stored as (2,2) arrays and I had no set method to perform the same process on a multi-dimensional level.
2. I could get a "mini-square" just fine, but how could I identify a particular square with the rows or columns that combine to make that square? (The picture below represents the dilemma, the red numbers identify the particular square, the orange numbers, the row and columns)




If you get to square 5, how do you know to look at rows 4, 5 and 6, or if you get to square 6, how do you know to look at rows 4, 5 and 6 and columns 7, 8 and 9.

How I chose to solve it:

1. I converted as much as I could from the 2 dimensional array, to a 1 dimensional array (except for the indices array, of possible locations where the number could be). Then, if a solution was found, I converted it back to a 2 dimensional array to get the correct location.
2. I had a switch statement at the beginning of the method to identify the "mini-square" that was being analyzed, and I ended up simply including the difference to add to the row and column for each "mini-square." For example, for square 8, as seen above, I would add 3 to the columns used, and 6 to the rows used. So if it was the 2nd column, 2nd row in the "mini-square", I would add 3 and 6 respectively to know to look at column 5, row 8 of the puzzle as a whole. The same method was applied to putting the correct number in the correct spot (as my method to do this needed a row, column and correct number as inputs).

## Summery

While this program is not capable of solving every Sudoku out there, it does a pretty good job at solving the large majority of Sudoku puzzles. A user, familiar with how this program fills in numbers, could use it to solve puzzles quicker, and get help with starting various puzzles.

Because of the way this program was designed, there may be a few additional techniques that could be added, but not too many—not without a redesign at the fundamental level.

After completing this program, if I had to do it again, I would strongly consider tackling it from a different angle, which might be more robust to adding future solving methods. I would probably assign a simple array to each cell, which contains all the numbers that might be had in that cell. With that knowledge stored between iterations, I think more powerful solving methods could be incorporated, more easily.

Thank you, I hope you enjoy this product

Taylor Sandbakken