

Final Project Report

Part 1:

Our final project that we decided to research on is the concept of a Sudoku puzzle solver. If you have never seen a Sudoku puzzle here is a template of one:

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

There are three key concepts/rules to follow when solving a Sudoku puzzle:

1. Each row contains distinct numbers from 1 to the width of the puzzle.
2. Each column contains distinct numbers from 1 to the width of the puzzle.
3. Each region (in our case above, each 3x3 square) has numbers 1 to the width.

If all of these rules are satisfied and the boxes within the puzzle are all filled, then the puzzle is solved. Here is the above picture solved:

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

Part 2:

Other than noticing that how our brute force method takes forever to complete, Takayuki Yato proved in 2003 that a solver to Number Place (Sudoku) is an NP-complete algorithm. Where Takayuki uses a reduction proof to prove Number place problems are NP-complete.

Part 3:

Our Sudoku project has little to no real life applications, it is just a puzzle solver that attempts to solve puzzles that people do for fun. The only ties to real world application is that if you find a P solution to these puzzles, one you win 1 million dollars and two you can then attempt to say all NP problems can be solved in P time, which is life breaking, Many of our everyday security algorithms rely on an NP process to solve in order to keep secrets, secret. A proof of polynomial time could be used to solve Latin Squares in polynomial time, which in turn gives solution to tri-partite graphs into triangles, which then can be used to find solutions to a special case of SAT known as 3-SAT, which lastly gives a solution to Boolean satisfiability. This gives the solution to solve any other NP problem in polynomial time.

Part 4:

The approximation algorithm we used is based on the following pseudocode:

1. Enumerate all empty cells in typewriter order (left to right, top to bottom)
2. Our “current cell” is the first cell in the enumeration.
3. Enter a 1 into the current cell. If this violates the Sudoku condition, try entering a 2, then a 3, and so forth, until
 - a. the entry does not violate the Sudoku condition, or until
 - b. you have reached 9 and still violate the Sudoku condition.
4. In case a: if the current cell was the last enumerated one, then the puzzle is solved. If not, then go back to step 2 with the “current cell” being the next cell.
 In case b: if the current cell is the first cell in the enumeration, then the Sudoku puzzle does not have a solution. If not, then erase the 9 from the current cell, call the previous cell in the enumeration the new “current cell”, and continue with step 3.

The next three images show how we implemented this algorithm in C#.

```

cs361_FinalProject - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug - Any CPU Start
Backtracker.cs PuzzleDatabase.cs GenFunctions.cs SudokuPuzzles Program.cs
cs361_FinalProject cs361_FinalProject.SudokuPuzzle SimpleSolver0

244 public string[,] SimpleSolver()
245 {
246     int width = puzzle.GetLength(0) - 1;
247
248     string[,] result = new string[width + 1, width + 1];
249
250     for (int i = 1; i < puzzle.GetLength(0); i++)
251     {
252         for (int j = 1; j < puzzle.GetLength(1); j++)
253         {
254             result[i, j] = puzzle[i, j];
255         }
256     }
257     Stack<Backtracker> backtrackers = new Stack<Backtracker>();
258
259     string temp1 = null;
260
261     string temp2 = null;
262
263     int counter = 0;
264
265     for (int i = 1; i < result.GetLength(0); i++)
266     {
267         for (int j = 1; j < result.GetLength(1); j++)
268         {
269
270             if (result[i, j] == null)
271             {
272                 int[] numbers = RandNumbers(width);
273
274                 for (int k = 0; k < numbers.Length; k++)
275                 {
276                     if (IsValidValue(result, i, j, numbers[k]) && numbers[k].ToString() != temp1 && numbers[k].ToString() != temp2)
277                     {
278                         result[i, j] = numbers[k].ToString();
279
280                         temp2 = null;
281
282                         temp1 = null;
283
284                         Backtracker backtracker = new Backtracker(i, j, numbers[k].ToString());
285
286                         backtrackers.Push(backtracker);
287
288                         break;
289                     }
290                 }
291                 if (result[i, j] == null)
292                 {
293                     if (counter == 1)
294                     {
295                         Backtracker backtracker1 = backtrackers.Pop();
296
297                         result[backtracker1.GetRow(), backtracker1.GetColumn()] = null;
298
299                         i = backtracker1.GetRow();
300
301                         Console.WriteLine "[" + i + ", " + j + "]";
302
303                         j = backtracker1.GetColumn() - 1;
304
305                         temp2 = backtracker1.GetValue();
306
307                         counter--;
308
309                         continue;
310                     }
311                     Backtracker backtracker2 = backtrackers.Pop();
312
313                     result[backtracker2.GetRow(), backtracker2.GetColumn()] = null;
314
315                     i = backtracker2.GetRow();
316
317                     Console.WriteLine "[" + i + ", " + j + "]";
318
319                     j = backtracker2.GetColumn() - 1;
320
321                     temp1 = backtracker2.GetValue();
322
323                     counter++;
324                 }
325                 numbers = RandNumbers(width);
326             }
327         }
328     }
329     return result;
330 }

```

First, we get the width of the puzzle that we grabbed from our pseudo-database. Each puzzle is 10 by 10, so that way we can index from 1 to 9. Thus, we need to subtract 1 from the length of the puzzle. Then we create a new 10 by 10 array of strings, and copy each value from the puzzle into it. Then we create a stack of backtrackers. A backtracker is just an object that holds on to a row index, a column index, and a string value. After that, we create 2 temp strings, and counter.

Next, we loop through the entire 2D array, and check if each value is null. If it is not, we go to the next value. Otherwise, we create an int array that holds the numbers between 1 and the width, inclusive, in a random order. Then we enter another loop from 0 to width, and check if that value can be placed in the current location, and it is not the same as either temp value. The isValidValue method will be explained later, and the reason we make sure the current value is not the same as either temp is for backtracking purposes. We keep looping until we find a value that fits, or find that no values fit. If we find one that fits, we store it in the 2D array, set both temps to null, and create a new backtracker that holds the value we just placed and its position. Then we break.

If we did not find a value to place, then we need to backtrack because we made a mistake earlier. If the counter created earlier is 1, then we pop the backtracker off the stack, set the value at the location of that backtracker to null, and set i and j to that location. We also set the second temp value to the value of the backtracker, so we can make sure we don't place the same value in that spot again. The counter is then decremented, and we hit a continue to break out of that if statement.

If, instead, the counter was 0, then we do the same thing, except this time we set the first temp value to the value of the backtracker, and we increment the counter. The reason we have both temp values is to make sure that if there were 2 seemingly valid numbers that can be placed in a spot, but both of them are actually wrong, we want to make sure we can see the correct value.

When the i and j values stop being set to smaller values in the backtrack process, the outer loops will cease, and the completed puzzle will be returned. The only issue is it does not work. We spent hours debugging this code but to no avail.

Part 5:

So what we should have seen if we were to finish the approximation code was that it would solve any 9x9 puzzle we threw at it in a decent amount of time and because that it would find a solution, the algorithm would be good enough. We did find a source that wrote a solver in C# with some very fancy usage of C# that we just don't have the background of C# to be able to write ourselves. At the end of this C# tutorial the author mentions another Sudoku solver coder who claims to have a solver that can take in 1000 hard level Sudoku puzzles and solve them all in less than 2 seconds. There will be a reference to the code and the solver webpage

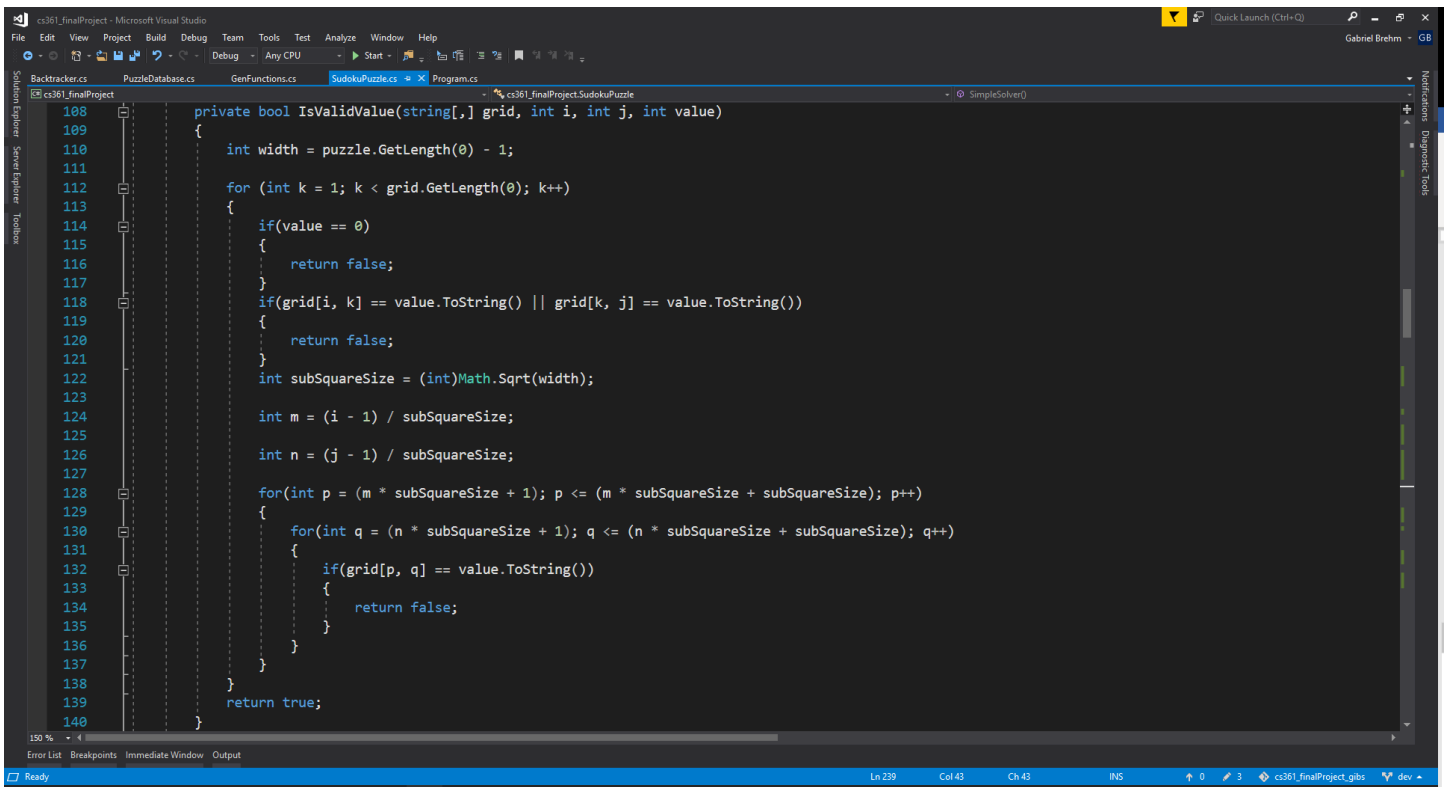
in our reference page, but no formal reference because we could not find any kind of formal publication of his work.

Part 6:

Well considering we didn't finish our approximation algorithm we can only assume that it is $O(a \text{ lot})$. We have found examples where it can seem that the complexity is pretty small, but these sources are only dealing with a 9x9 grids as soon as you go up to 16x16, those algorithms don't work nearly as fast.

Our Approach

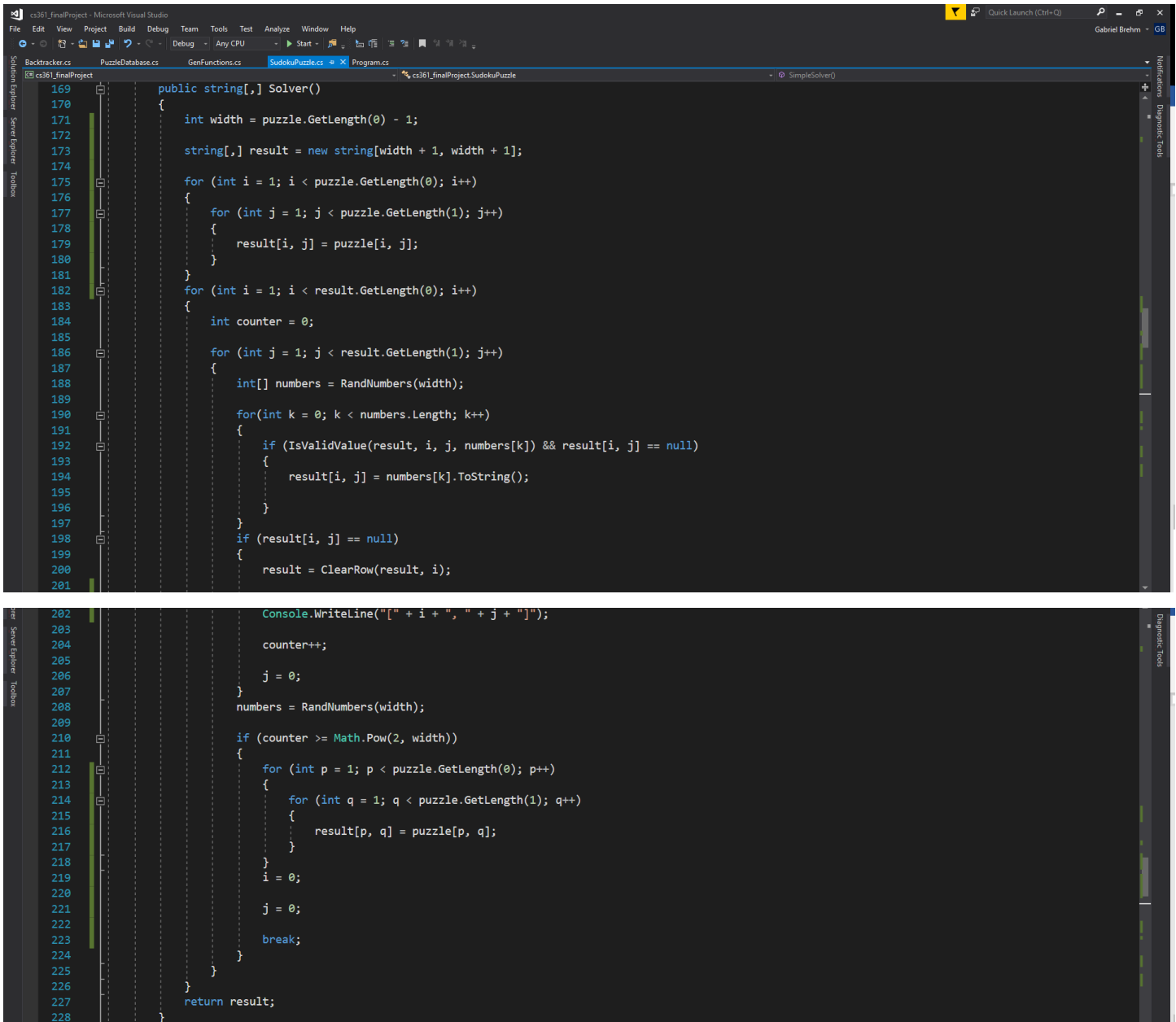
Our approximation algorithm will follow. It is not glamorous, but it has yet to fail solving a 9 by 9 puzzle. It just might take a long time. The backbone of both our algorithm, and the one described above, is the `isValidValue` method. An image of it can be seen below.



This method takes in the puzzle grid we are working with, the row index, the column index, and the value we want to place there. For the same reason as before, the width is set to the grid's width minus 1. Then we loop from `k` equals 0 to the grid's width, checking for 3 different conditions to be true. The first is that the value is not 0; if it is, we return false. Second, we check to make sure that the row and column of that index do not contain the value; if either does, we return false. Lastly, we need to make sure that the square that the index is in does not contain the value. This is done by determining the size of each of the sub squares of the puzzle, then we set variables `m` and `n` to the row and column indices, respectively, minus 1 divided by the sub square size. This will give us the "index" of the whole sub square. Once we know that, we just loop through

those values and determine if any of them equal the given value; if one does, we return false. If none of those conditions are met, we return true.

Now we will discuss our approximation algorithm, pictured below. It is pretty much a brute force algorithm. It just tries to solve the puzzle adhering to the rules, and if it spends too long repeatedly resetting a row, it just starts over. It works based on the idea that it won't try the same numbers every time, so eventually, it will stumble upon the correct solution.



```

169 public string[,] Solver()
170 {
171     int width = puzzle.GetLength(0) - 1;
172     string[,] result = new string[width + 1, width + 1];
173
174     for (int i = 1; i < puzzle.GetLength(0); i++)
175     {
176         for (int j = 1; j < puzzle.GetLength(1); j++)
177         {
178             result[i, j] = puzzle[i, j];
179         }
180     }
181     for (int i = 1; i < result.GetLength(0); i++)
182     {
183         int counter = 0;
184
185         for (int j = 1; j < result.GetLength(1); j++)
186         {
187             int[] numbers = RandNumbers(width);
188
189             for (int k = 0; k < numbers.Length; k++)
190             {
191                 if (IsValidValue(result, i, j, numbers[k]) && result[i, j] == null)
192                 {
193                     result[i, j] = numbers[k].ToString();
194                 }
195             }
196             if (result[i, j] == null)
197             {
198                 result = ClearRow(result, i);
199             }
200         }
201
202         Console.WriteLine("i = " + i + ", j = " + j);
203         counter++;
204         j = 0;
205     }
206     numbers = RandNumbers(width);
207
208     if (counter >= Math.Pow(2, width))
209     {
210         for (int p = 1; p < puzzle.GetLength(0); p++)
211         {
212             for (int q = 1; q < puzzle.GetLength(1); q++)
213             {
214                 result[p, q] = puzzle[p, q];
215             }
216             i = 0;
217             j = 0;
218             break;
219         }
220     }
221     return result;
222 }
223
224 private void ClearRow(string[,] result, int i)
225 {
226     for (int j = 1; j < result.GetLength(1); j++)
227     {
228         result[i, j] = null;
229     }
230 }

```

First, we set the width the same way we have been. Then we make a 2D string array, and make a copy of the puzzle. Then we loop through each row of the puzzle. The first thing we do in there, is create a counter and set it equal to 0. Then we loop through each column, in which, we start by creating an int array that holds the numbers between 1 and the width, inclusive, in a random order. Then we enter another loop from $k = 0$ to the

width, and try to assign each value in the array of random numbers to the puzzle, given that the location was null, and the isValidValue method returned true. If none of the values could be placed, we call the ClearRow method, which just resets the values in that row to their originals. We then increment that counter, and set j to 0.

After that, the array of random numbers is reset to a new array of random numbers. Then, if the counter is greater than 2 to the width, we reset the copy of the puzzle back to the original puzzle, and set i and j to 0, then break. Through testing, we determined that 2 to the width was a sort of happy medium. It is not so small that the algorithm will hit it before making any headway of solving the puzzle, but it is not so big that it just sits somewhere stuck for too long.

Code Outputs:

```

C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\cs361_finalProject_gibs...
[9x9 Sudoku Puzzle]
+-----+
| 8 | 4 |  | 1 | 2 |  | 6 | 9 |  |
+-----+
|  |  |  | 3 |  |  | 4 |  | 2 |
+-----+
|  | 2 | 3 |  |  |  |  | 5 |  |
+-----+
|  | 5 |  |  | 2 | 6 |  |  |  |
+-----+
| 6 |  |  | 4 | 9 | 7 |  |  | 5 |
+-----+
|  | 1 | 5 |  |  | 2 |  |  |  |
+-----+
|  | 7 |  |  |  | 9 | 8 |  |  |
+-----+
| 2 |  |  |  | 3 |  |  |  |  |
+-----+
|  | 9 | 6 |  | 4 | 8 |  | 3 | 1 |
+-----+

[SOLUTION: 9x9 Sudoku Puzzle]
+-----+
| 8 | 4 | 7 | 1 | 2 | 5 | 6 | 9 | 3 |
+-----+
| 1 | 6 | 5 | 3 | 8 | 9 | 4 | 7 | 2 |
+-----+
| 9 | 2 | 3 | 6 | 7 | 4 | 1 | 5 | 8 |
+-----+
| 7 | 5 | 9 | 8 | 1 | 2 | 3 | 6 | 4 |
+-----+
| 6 | 3 | 2 | 4 | 9 | 7 | 8 | 1 | 5 |
+-----+
| 4 | 1 | 8 | 5 | 3 | 6 | 7 | 2 | 9 |
+-----+
| 3 | 7 | 4 | 2 | 5 | 1 | 9 | 8 | 6 |
+-----+
| 2 | 8 | 1 | 9 | 6 | 3 | 5 | 4 | 7 |
+-----+
| 5 | 9 | 6 | 7 | 4 | 8 | 2 | 3 | 1 |
+-----+

+-----+
| Press ENTER to EXIT |
+-----+

C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\cs361_finalProject_gibs...
[9x9 Sudoku Puzzle]
+-----+
|  |  |  |  | 4 |  |  | 2 |  |
+-----+
| 3 |  |  | 9 |  | 6 | 4 | 8 |  |
+-----+
|  | 5 | 8 |  | 9 |  | 1 |  |  |
+-----+
|  | 6 |  |  | 3 | 7 |  |  |  |
+-----+
| 7 | 4 | 9 |  |  | 3 | 2 | 6 |  |
+-----+
|  | 2 | 6 |  |  | 8 |  |  |  |
+-----+
| 9 | 7 |  |  | 2 |  | 6 |  |  |
+-----+
| 5 | 2 | 3 |  | 8 |  | 4 |  |  |
+-----+
| 4 |  | 1 |  |  |  |  |  |  |
+-----+

[SOLUTION: 9x9 Sudoku Puzzle]
+-----+
| 6 | 9 | 8 | 3 | 1 | 4 | 5 | 7 | 2 |
+-----+
| 3 | 7 | 1 | 9 | 2 | 5 | 6 | 4 | 8 |
+-----+
| 2 | 5 | 4 | 8 | 7 | 6 | 9 | 3 | 1 |
+-----+
| 8 | 6 | 5 | 2 | 4 | 3 | 7 | 1 | 9 |
+-----+
| 7 | 4 | 9 | 5 | 8 | 1 | 3 | 2 | 6 |
+-----+
| 1 | 3 | 2 | 6 | 9 | 7 | 4 | 8 | 5 |
+-----+
| 9 | 1 | 7 | 4 | 5 | 2 | 8 | 6 | 3 |
+-----+
| 5 | 2 | 3 | 7 | 6 | 8 | 1 | 9 | 4 |
+-----+
| 4 | 8 | 6 | 1 | 3 | 9 | 2 | 5 | 7 |
+-----+

+-----+
| Press ENTER to EXIT |
+-----+

```

References:

Mathematics and Sudokus. (2009, Summer). Retrieved June 10, 2019, from http://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Solving_any_Sudoku_I.html

Simplicity, B. (2016, August 18). Sudoku Design and Implementation in C#. Retrieved June 14, 2019, from <http://brutalsimplicity.github.io/2016/08/18/sudoku.html>

Yato, T. (2003, January). Complexity and completeness of Finding Another Solution and its Application to Puzzles. Retrieved from <http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf>

Fastest Solver (we could find):

<https://attractivechaos.github.io/plb/kudoku.html>

The code:

https://raw.githubusercontent.com/attractivechaos/plb/master/sudoku/sudoku_v1.c