Bits & Bytes

Arkansas' Premier Computer Club

June 2020

Bella Vista Computer Club - John Ruehle Center

Highlands Crossing Center, 1801 Forest Hills Blvd Suite 208 (lower level), Bella Vista, AR 72715

Website: http://BVComputerClub.org **Email: editor@bvcomputerclub.org**

COVID-19 VIRUS ADJUSTMENTS

During normal times all meetings are on the lower level of the Highlands Crossing Center in Bella Vista. For the month of June we will continue to suspend in-person meetings and classes and conduct on-line meetings using Zoom over the Internet.

To attend a Zoom meeting or class, you need Internet access and a device with the Zoom application installed.

MEETINGS

(Online) Board Meeting: June 8, 6pm, using Zoom

(Online) General Meeting: June 8, 7pm, "**A Look At Google Docs**" Zoom meeting access information will be emailed to membership the week before. Visitors or Guests may contact our Membership Chair for Zoom meeting details at membership@bvcomputerclub.org

Genealogy SIG: **CANCELED for June.** (3rd Saturday).

HELP CLINICS

No June Help Clinics at John Ruehle center

Members may request Remote Help on our website at https://bvcomputerclub.org at menu path Member Benefits ►Remote Help .

MEMBERSHIP

Single membership is $25; $10 for each additional family member in the same household. Join by mailing an application (from the web site) with check, or complete an application and pay at any meeting.

It is now also possible to Join or Renew membership on line on our website at https://bvcomputercub.org at menu path Get Involved ►Join/Renew . Payment may be by Credit Card, or, if you have a PayPal account, by whatever means you have defined on PayPal.

CLASSES

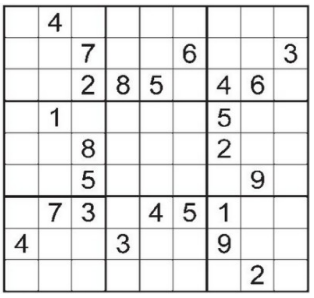
(Online) "Using Windows 10" – Joel Ewing, Wednesday, June 24, 9am - noon.

Advance sign up required for each listed class: Contact Grace: email to edu@bvcomputerclub.org, text 469-733-8395, call 479-270-1643, or sign up at the General Meeting. Classes are **free to Computer Club members.** Class access information will be emailed to those signed up for the class the day before class.

Check the monthly calendar and announcements for any last minute schedule changes at http://bvcomputerclub.org .

HOW TO TEACH A COMPUTER TO SOLVE SUDOKU PUZZLES

By Joel Ewing
President, Bella Vista Computer Club
President (at) bvcomputerclub.org



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

I first became aware of Sudoku puzzles several decades back when several of my co-workers were regularly solving the puzzles in the morning paper. The classic Sudoku puzzle is a 9x9 grid of cells, each one of which can contain one of the digits 1 to 9, further sub-divided into nine 3x3 grids of cells. Given a starting grid, like the one to the right, the object is to fill in all the empty cells so that each row, each column, and each 3x3 grid contains all the numbers 1 to 9 in some order. The published puzzles are supposed to have one unique solution, but you could easily devise a puzzle variant in which multiple solutions are possible (by just omitting a number from the original puzzle).

There are no doubt many books or articles available by now about solving Sudoku puzzles, but I preferred finding my own approach. Not being sure at first whether there would always be a unique solution I tried to think of how an automated approach might find all possible solutions. I will attempt to describe the general algorithm used for an automated approach without going into the implementation details. These techniques can also be applied to a manual solution, but may be tedious.

The first step is to come up with a process, or "algorithm", that is guaranteed to find a solution, or terminate if none exists, and which is practical.

An example of a process that will clearly work and eventually terminate is an exhaustive search: take the first empty cell, create 9 new puzzles, each with a different value 1 to 9 in that cell; if all the cells are filled, check if it obeys the rules: if so, a solution is found, if not discard. If some cells are still empty, then try to solve the resulting puzzle by the same technique. For the example puzzle, this technique results in 9^{57} possible board combinations to test, which is MANY times the age of the Earth in nanoseconds, so it clearly is not practical.

A much more efficient process can be created by eliminating any cell value choices at each step that clearly cannot lead to a solution. If the first cell chosen to fill is the upper-left cell (row 1, column 1), then values already in the same row, column, and box immediately eliminate choosing 2, 4, or 7 for cell (1,1), so the possible candidates for cell (1,1) are among the digits 135689 . Whichever of those values we choose to try for cell (1,1), potentially reduces the possibilities for all other empty cells in the same row, column, and box. There is no easy way to estimate the number of choices involved to reach a solution, but there is reason to hope this would lead to a practical solution. This is still an exhaustive search, just with elimination of unproductive paths as early as possible.

When solving a Sudoku puzzle by hand, you don't process the empty cells in order, but try to select cells to work on by intuition and recognizing patterns. In the above process, the order of choosing empty cells to fill is not important as long as it is well-defined. Starting with cell (1,1) may not be a good choice: in the worst case we might choose 5 values that didn't work before succeeding on the last. Let's improve the odds by finding a cell that has fewer possible choices.

While it is tedious to determine by hand the candidate list for each empty cell, for a program this is trivial once the process is defined for a single cell. Figure 2 below shows the initial status for the example puzzle, where "0" followed by a string of digits is a candidate list for that cell, and a single non-zero digit is the value already set for a cell. Note that the candidate list for cell (1,1) is indeed "135689" as previously discussed.

0135689	4	0169	01279	012379	012379	078	01578	0125789
01589	0589	7	01249	0129	6	08	0158	3
0139	039	2	8	5	01379	4	6	0179
023679	1	0469	024679	0236789	0234789	5	03478	04678
03679	0369	8	0145679	013679	013479	2	01347	01467
02367	0236	5	012467	0123678	0123478	03678	9	014678
02689	7	3	0269	4	5	1	08	068
4	02568	016	3	012678	01278	9	0578	05678
015689	05689	0169	01679	016789	01789	03678	2	045678

Figure 2

One pattern should immediately stand out. Note that for some cells, namely (2,7) and (7,8), there is only a single candidate ("08" in both cases). If there is a solution, clearly any cell with only one possible value must be assigned that value. So, modify the process to first handle all the cells with a single candidate, assign that value to the cell and re-compute the candidate lists for all empty cells in the same row, column and sub-grid, producing in this case:

0135689	4	0169	01279	012379	012379	07	0157	012579
0159	059	7	01249	0129	6	8	015	3
0139	039	2	8	5	01379	4	6	0179
023679	1	0469	024679	0236789	0234789	5	0347	04678
03679	0369	8	0145679	013679	013479	2	01347	01467
02367	0236	5	012467	0123678	0123478	0367	9	014678
0269	7	3	0269	4	5	1	8	06
4	02568	016	3	012678	01278	9	057	0567
015689	05689	0169	01679	016789	01789	0367	2	04567

Figure 3

Notice we now have new cells with only a single candidate left: "07" at (1,7) and "06" at (7,9). An obvious thing to try is to keep applying this "1-candidate" strategy in the hope of eventually reaching a solution, but with this puzzle we eventually reach a point where no cell has a single candidate:

0135689	4	0169	0129	01239	01239	7	015	01259
0159	059	7	01249	0129	6	8	015	3
0139	039	2	8	5	01379	4	6	019
023679	1	0469	024679	0236789	0234789	5	0347	0478
03679	0369	8	0145679	013679	013479	2	01347	0147
0237	023	5	01247	012378	0123478	6	9	01478
029	7	3	029	4	5	1	8	6
4	02568	016	3	012678	01278	9	057	057
015689	05689	0169	01679	016789	01789	3	2	0457

Figure 4

The simplest logical way to proceed is to revert to a modified form of the original process: when there is no 1-candidate cell, pick a cell with the smallest number of candidates (there are several above with only two choices), create a new Sudoku puzzle from the currently partially solved puzzle trying each of the candidate values in turn for that cell, and see which of the resulting simpler puzzles has a solution. If at some point that choice results in a puzzle state where any cell has a candidate list of "0" (no valid choices), that indicates the last chosen value produces no solution. This is a concept called "recursion", where a process performing some computation may at some point re-invoke itself to perform a simpler computation as part of the first computation. It may progress to many different levels as long as the computation gets simpler at each recursion, so that eventually an answer (there "is" or "is not" a solution) can be produced that bubbles back up the chain of invocations. This sounds like a complex process, but many different programming languages support recursion in some form and in this case only minimal additional work was required to implement it.

It should be apparent that the above process is guaranteed to find a solution, or as many solutions as might exist. At each step either a cell value is set because there is no other choice, or when multiple choices are possible, all choices are tried. Since these rules tell explicitly how to proceed at each step, this is indeed the kind of process that can be automated.

Implementing the above algorithm still involves some art, as there is an arbitrary choice of programming language, which in turn influences the choice of how to represent the Sudoku board and all the row, column, and sub-grid relationships and how to communicate the initial Board setup to the program.

The language used for the current implementation is ooREXX (Open Object REXX), which is one of many free languages available on Linux and also available for Windows. My first Sudoku program version was written in Regina Rexx under Windows and Linux, and required only a few minor modifications to run under ooREXX (or under REXX on IBM mainframe computers under z/OS).

Simple lines of text are used for both input and output. I wanted to expend effort on the actual puzzle solution logic, not on supporting fancy graphic input and output. The puzzle is entered as one line for each initially-filled cell, in the format of "value row-column", as in 4 12, 7 23, 6 26, 3 29 …

The board itself is represented as an array with 81 elements whose values are a combination of cell values and candidate lists (Figure 2 is a an example of printing that array just after the initial puzzle definition). Rows, columns, and sub-grids all act alike – as a group of related cells where each cell within the group must have a different value from other cells in the same group – so it makes sense to represent them all as a "group" of cells. The 9 rows, 9 columns, and 9 3x3 grids are represented by an array of 27 groups, each of which lists the 9 cells contained in the group. In addition there is another 81-element array corresponding to each of the cells on the board, which lists for each cell to what groups the cell belongs. This structure makes it possible for the program to know, whenever a cell value is set, what groups (row, column, and sub-grid) it affects, and from that which other cells must have a candidate list adjusted.

The actual "Solve" process starts at "level 0", with higher levels only required when a trial guess with a recursive call to Solve a simpler Sudoku puzzle is employed.

For the example puzzle above, the program found a unique solution in 1/20 of a second as

```
   A Solution:
 8 4 6 1 3 9 7 5 2
 9 5 7 4 2 6 8 1 3
 1 3 2 8 5 7 4 6 9
 6 1 4 2 9 8 5 3 7
 7 9 8 5 6 3 2 4 1
 3 2 5 7 1 4 6 9 8
 2 7 3 9 4 5 1 8 6
 4 6 1 3 8 2 9 7 5
 5 8 9 6 7 1 3 2 4
successes 1
trials  5
fail    1
Forced 1 Cand moves at level 0:  6
Forced 1 Cand moves at higher level:  62
real  0m0.048s
```

This solution required 5 trial guesses, 1 of which produced no solution. After finding this solution, it proceeded to try other candidate choices for a total of 8 trials and 4 failures, and found no additional solutions. The execution time values were achieved by building data into the program to eliminate typing time.

Improving The Algorithm

After writing this program I was occasionally tempted to solve the daily puzzle by hand and found that one learns to use various visual clues and a combination of other techniques that would be very hard to describe computationally. For example, one tends to focus first on more frequently occurring values and on rows, columns, or 3x3 boxes with the fewest empty cells. Devising notations to track if the placement for a value within a box was reduced to two cells, a single column, or single row frequently made it possible to deduce additional information in other boxes. It became apparent that in most if not all of the newspaper puzzles, it was never necessary to use trial and error to arrive at a solution, which suggested it might be possible to eliminate the need for trial values and recursion in the computer algorithm as well. The motivation for this was intellectual curiosity, not a need for the computer program to actually run faster, as it typically only took 60 seconds to arrive at a solution and all but the last 1/20 second of that was just typing in the puzzle definition.

One technique I found myself using by hand when the number of empty cells in a row, column, or box got down to 4 or less is a technique I'll call a "value-based" move; namely, for each of the missing values in the group, check to see if there is one and only one possible location for that value, which means that cell must be assigned that value. It turns out the cell candidate lists already created for the 1-candidate moves can also be efficiently used to look for possible value-based moves.

Looking at Figure 4, where there were no remaining 1-candidate moves, look at cell (1,9) which has a candidate list of "1 2 5 9". Next look at the candidate lists for all the other cells in the upper-right 3x3 box. The only other empty cells in that box have candidate lists of "1 5" , "1 5" and "1 9". Cell (1,9) is thus the only possible location for the value "2" within that box, so that cell must contain a "2". Clearly adding "value-based" move support would allow additional progress to be made before it would be necessary to resort to trial guesses and recursion,

Support for value-based moves to check ALL rows, columns, and boxes for possible moves was added to the Sudoku program. This also would be a pain to do by hand – unless desperate, one usually only tries groups with a small number of empty cells, but doing all groups is actually easier to code for a computer than trying to determine which groups might be the best choices. Initially I wanted to find out how often the value-based technique was needed to bail out the 1-candidate based technique, so it only looked for one value-based move before resuming a search for 1-candidate moves, resorting to the candidate trial and recursion only if neither of those other methods could advance.

Adding that support produced the same solution but with the following statistics:

```
successes 1
trials  0
fail    0
Forced 1 Cand moves at level 0:  53
Forced 1 Cand moves at higher level:  0
Forced Value moves at level 0:  4
Forced Value moves at higher level:  0
real  0m0.045s
```

No guesses (trials) or recursion was needed to produce this solution, and only 4 times was a value-move needed to bail out the 1-candidate move process.

Out of curiosity, I tried one more solution process variant: Start using value-based moves and see how far that would get before having to use a 1-candidate move to bail out that technique. This produced an unanticipated result:

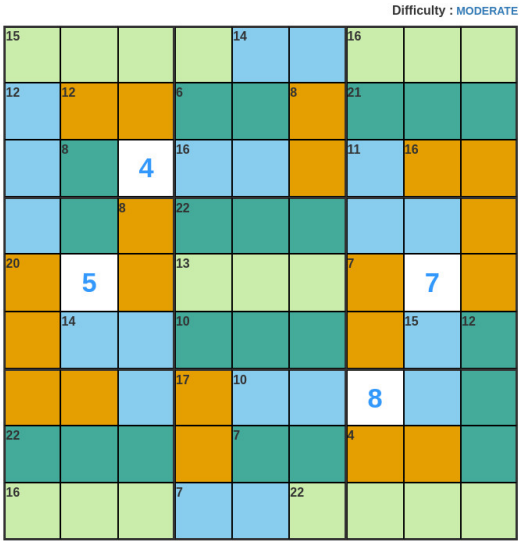
```
successes 1
trials  0
fail    0
Forced 1 Cand moves at level 0:  0
Forced 1 Cand moves at higher level:  0
Forced Value moves at level 0:  57
Forced Value moves at higher level:  0
real  0m0.049s
```

All moves required for a solution were done using just the value-move technique! The next 8 daily Sudoku puzzles tried had a similar result. Finally on the 9th puzzle tried, I found a puzzle where there were four instances during the solution process where no value-based move was possible but a 1-candidate-based move could be made to avoid having to make a trial guess and use recursion to continue. It is unclear whether the value-based technique is on average more effective, or if the newspaper puzzles are just designed in a way that improves the odds they can be solved using only the value-based technique. In either case, whenever stumped trying to solve those puzzles manually, odds favor there being a value-based move that has been overlooked.

Sudoku Extensions

Since the NWA Democrat-Gazette is planning to stop hard-copy home delivery of the weekday paper later in 2020, I've been trying to read the online digital version more. The thing that got me to thinking about Sudoku solutions recently was stumbling across a more difficult Sudoku variant, Killer Sudoku, in the online version of

the NWADG newspaper, and wondering how difficult it would be to modify my old Sudoku program to solve those as well. See the Killer Sudoku example to the right. This variant has the same row, column, and sub-grid rules as regular Sudoku. It specifies many fewer cells, and then defines sum-groups of contiguous cells (represented by the different colors) that must add up to some specified number (the small value in the upper left corner of one of the cells), while also requiring all the values within the sum-group also be unique. The additional constraints imposed by the summation groups give a unique solution with only a few specified cell values.



In the example above, the two-cell group starting at row 2 column 2 or (2,2) must add up to 12, which means the two cells must have the values (in any order) of 3,9 4,8 or 5,7. Note that 6,6 is disallowed because that would use the same digit twice in the same sum-group. That means that initially this sum-group adds a candidate restriction of 345789 to those two cells in addition to the candidate list imposed by regular Sudoku rules, so (2,2) which would in normal Sudoku have possible candidates of 12346789 would have to remove those candidates not also in the new sum-group candidate list, becoming 34789. Similarly the candidate list for (2,3) of 12356789 modified for the sum-group would become 35789. Once either cell (2,2) or (2,3) is assigned a value, that will reduce the candidate list for the other cell in the that sum-group to just one choice. With sum-groups, every time a cell in the group is assigned a value, the possibilities for the remaining cells in the sum-group must be recalculated. That's trivial for a two-cell sum-group, but not so for groups with more cells. The most difficult part of this extension is coming up with an algorithm to calculate a sum-group candidate list for the remaining cells in a sum-group.

Sum-groups are entered to the program in the format of "sumvalue row-column1 row-column2 … ", so that the sum-group starting at (2,2) would be entered as "12 22 23". Entering the definition for the above puzzle is rather tedious as there are 26 different sum-groups and 77 different cell references to type, and you have to carefully think about row and column numbers while typing.

With few initial cells specified, you are less likely to have cells with only a single candidate value or values that have a single placement within a group, so the candidate trial method and recursion should be needed. Finding a solution took noticeably longer, but still under 1 second for the computed solution for the above puzzle. Exhausting 1-candidate moves first before trying to find a value-based move gave the following result:

```
 2 7 5 1 6 8 4 3 9
 1 3 9 4 2 5 7 8 6
 8 6 4 7 9 3 1 5 2
 3 2 7 5 8 9 6 4 1
 9 5 1 3 4 6 2 7 8
 6 4 8 2 7 1 5 9 3
 4 1 2 9 3 7 8 6 5
 7 9 6 8 5 2 3 1 4
 5 8 3 6 1 4 9 2 7
successes 1
trials  45
fail    35
```

```
Forced 1 Cand moves at level 0:  6
Forced 1 Cand moves at higher level:  153
Forced Value moves at level 0:  0
Forced Value moves at higher level:  119
real  0m0.650s
```

Changing parameters to exhaust all value-based moves before looking for 1-candidate moves changed the stats to:

```
successes 1
trials  45
fail    35
Forced 1 Cand moves at level 0:  6
Forced 1 Cand moves at higher level:  59
Forced Value moves at level 0:  0
Forced Value moves at higher level:  265
real  0m0.966s
```

For reasons unclear, it appears preferring value-based moves performs less efficiently than preferring 1-candidate moves on Killer Sudoku puzzles (see the real time for solution), but in either case the 1-candidate technique, the value-based technique, and multiple-choice technique with recursion were all needed to arrive at the solution.

Maybe there are some possible shortcuts when doing Killer Sudoku puzzles by hand, but the apparent need for trial guesses would seem to make those puzzles much more tedious to solve manually, compared with a regular Sudoku puzzle!

There are many other variants of Sudoku. I believe all can be solved by similar approaches.
