Data Structures and Algorithm

Project Report of Sudoku Solver

SUBMITTED BY:

AKASH SRIVASTAVA

ROLL NO- MT2014004

M.TECH (INFORMATION TECHNOLOGY)

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY BANGALORE

How to solve sudoku puzzles

Solving sudoku puzzles is easier than it looks, and all but the very hardest puzzles can be solved using just a few simple techniques. We don't need any maths, and we don't need to guess. It's all done by observation and logic, and the most important thing is to stay 100% accurate at all times.
I have implemented five rules in my Sudoku code to met project requirements.

These five rules are follows.

1. Crosshatching
2. Rule 1 (Single-candidate squares)
3. Rule 2 (Single square candidate)
4. Pairs
5. Triples.

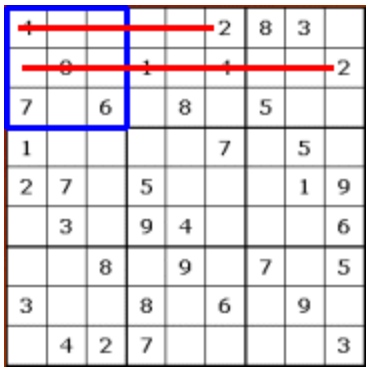
Crosshatching - finding squares for numbers

The obvious way to solve a sudoku puzzle is to find the right numbers to go in the squares. This uses a technique called 'crosshatching'. It can solve many 'easy' rated puzzles on its own.

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

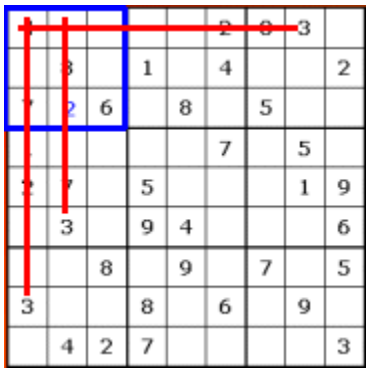
Crosshatching works in boxes (the 3 X 3 square subdivisions of the grid).We can see top-left box of our sample puzzle (outlined in blue). It has five empty squares. All the numbers from 1 to 9 must appear in the box, so the missing numbers are 1,2,3,5 and 9. We'll ignore 1 for a moment and see if we can work out which square the missing 2 will go into.

To do this, we'll use the fact that a number can only appear once in any row or column. We start by looking across the rows that run through this box, to see if any of them already contain a 2. Here's the result:



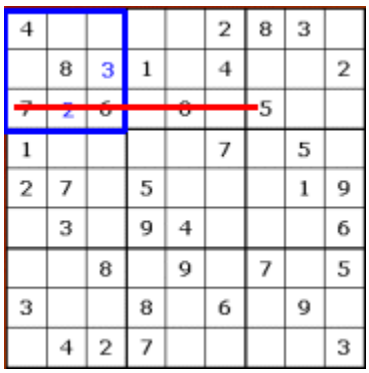
The first two rows already contain 2s, which means that squares in those rows can't possibly contain the 2 for this box. And the third row only has one empty square, so that must be the home for the 2.

Now see if we can place the 3 for this box. This time we end up checking the columns that run down through the box, as well as the rows that run across it:



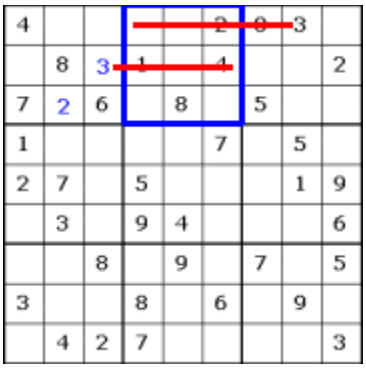
Again, we get a result first time - there's only one empty square that the 3 can possibly go into. You can see from this example why it's called 'crosshatching' - the lines from rows and columns outside the square criss-cross each other.

Of course we don't always get a result first time. Here's what happens when we try to place the 5:



There's only one 5 already in the rows and columns that run through this box. That leaves three empty squares as possible homes for the 5. For the time being, this box's 5 (and its 1 and 9) have to remain unsolved.

Now we move on to the next box:



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

Here we're crosshatching for 3, the first missing number in this box. Note how we treat the 3 we placed in the first box as if it had been pre-printed on the puzzle. We still can't place this box's 3 though, so we'll move on to the next missing number (5), and so on.

In sudoku, accuracy is essential. If the 3 in the first box is wrong, we'll be starting a chain of errors that may prove impossible to unravel. Only place a number when you can prove, logically, that it belongs there.**Never guess**, and never follow hunches!

An important factor in crosshatching (and sudoku in general) is that the more numbers you place, the more likely we are to place others - including ones we couldn't place earlier.

Placing numbers in the second box may well make it possible to go back and place missing numbers in the first. It's good to get into the habit of looking backwards as well as forwards, re-checking whether the numbers we've just placed have made numbers place-able elsewhere in the puzzle.

To implement this I have taken two dimensional Array for missing numbers for a particular box.

Now crosshatching is completed.To implement all other rule we need to do penciling-in.

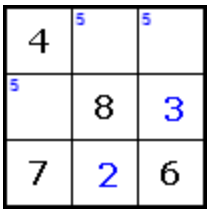
Algorithm: For this I have taken n*n matrix which stores the missing numbers of each box.

Now Traverse each cell of box if that missing number can place in that cell then that cell is the one possible candidate for that number,then go ahead and if another cell can take that number that it would not be filled in that cell. Otherwise fill the candidate if no other cell is the candidate for that number.

Pencilling in

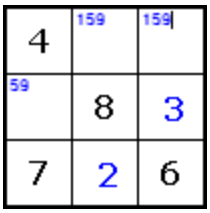
The solving techniques needed for more difficult puzzles all depend on having an accurate list of the possible numbers (called 'candidates') for each empty square. We can build this list by pencilling the candidates in as you make a complete crosshatching pass through the puzzle.

Looking at the top-left box of our original puzzle, crosshatching produced three possible squares where the missing 5 could go. Here's the box, with 5 'pencilled-in' to the corners of its three possible squares:



◊ (It's called 'pencilling-in' because on a printed puzzle many people use a pencil, so that they can rub numbers out later (an essential part of the solving process). On my sudoku page, you click in the top-left corner of the square, then type and delete numbers like a normal text box)

1 and 9 were also unsolved for this box. Here's the box with all its missing numbers pencilled into their possible squares:



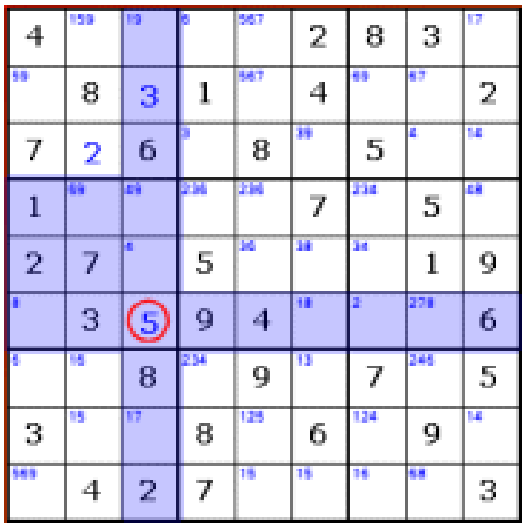
This list must be complete and accurate, otherwise you risk creating another chain of errors. That's why it's essential to crosshatch every missing number for every box before starting the next stage of solving.

Candidate lists must also be kept up to date (the reasons for this will become obvious later!). Here's what it means:

On the left is is the top-left box, and the one below it. We've just entered a 5 in the bottom-right square of the lower box.

Now we remove that square's candidate list. We also remove 5 from the candidate list at the top of the same column, and the left of the same row. Here's how the boxes look afterwards:

Whenever you fill in a square, remove the number you've used from all candidate lists in the same row, column and box. Here are the areas we needed to check for candidate 5s after filling in this square:



if that looks complex - in practice it's quick and easy to scan through the same row, column and box as the square we've just filled.
For this i used link list for each cell candidates.

Algorithm: I used the same missing number array to implement the candidate list of each cell.

For example: for cell one: 5->4->3->2

In the same all non filled cell will have possible candidate list.

Struct list{

int candidate;

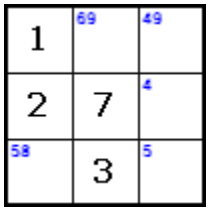
struct list *nextcandidate;

}

for the candidate count I have taken a seperate matrix which takes the candidate count value in the particular cell.

Rule 1 - Single-candidate squares.

When a square has just one candidate, that number goes into the square.
Here's the mid-left box again, as it was before we entered the 5:



The mid-right and bottom-right squares each have only one candidate, so we can put those numbers into the squares.

Some squares will be single-candidate from the start of the puzzle. Most, however, will start with multiple candidates and gradually reduce down to single-candidate status.

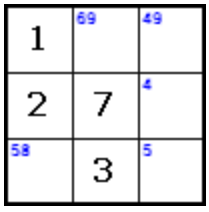
This will happen as we remove numbers that you've placed in other squares in the same row, column and box, and as we apply the last three rules described below.

Algorithm: if any cell candidate count is one assign that candidate to that cell and remove from other cell that is same row ,column,box.

Rule 2 - single-square candidates.

When a candidate number appears just once in an area (row, column or box), that number goes into the square.

Look at the mid-left box again:



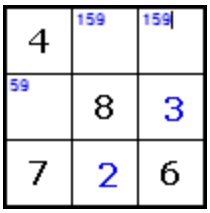
The number 6 only appears in one square's candidate list within this box (top-middle). This must, therefore, be the right place for the 6.

The remaining three rules let you remove numbers from candidate lists, reducing them down towards meeting one of the first two rules.

Rule 3 - number claiming.

When a candidate number only appears in one row or column of a box, the box 'claims' that number within the entire row or column.

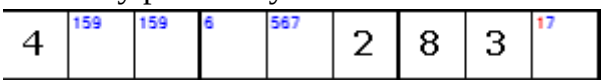
Here's the top-left box again:



The number 1 only appears as a candidate in the top row of the box. This means there will have to be a 1 somewhere in the first three squares of the puzzle's first row (i.e. the

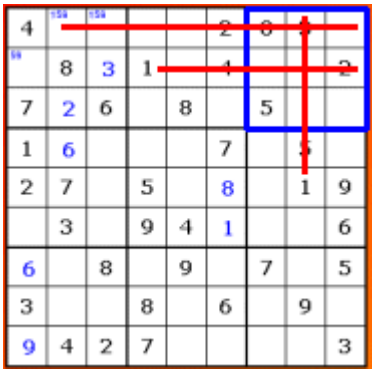
ones that overlap with the box). That in turn means that 1 can't go anywhere else in that row, outside of this box.

You can read across the row, and remove 1 from any candidate lists outside of this box, even though you haven't actually placed 1 yet.



In this example, we can remove the 1 from the right-hand square's candidate list. This makes the square single-candidate (7) - square solved!

Claims also work during crosshatching. Here we're crosshatching the top-right box for 1:

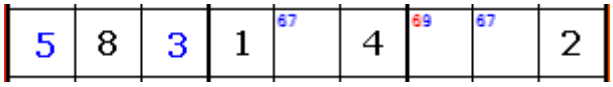


We can rule out the top row, because the top-left box has already claimed that row's 1. This lets us place the 1 in the bottom-right square of the box.

Rule 4 - pairs.

When two squares in the same area (row, column or box) have identical two-number candidate lists, you can remove both numbers from other candidate lists in that area.

Here's the second row of the puzzle:



Two of the squares have the same candidate list - 67. This means that between them, they will use up the 6 and 7 for this row.

That means that the other square can't possibly contain a 6. We can remove the 6 from its candidate list, leaving just 9 - square solved!

The squares in a pair must have exactly two candidates. If one of the above squares had been 679, it couldn't have been part of a pair.

Algorithm:

first check the candidate count if it is 2, then go to next cell in the box whoose candidate count is two then check the candidates of them if they are same remove them from all other candidate list.

Rule 5 - triples.

Three squares in an area (row, column or box) form a triple when:

- None of them has more than three candidates.

- Their candidate lists are all full or sub sets of the same three-candidate list (explained below!).

You can remove numbers that appear in the triple from other candidate lists in the same area.

Here's the fourth row of the puzzle:

Note the three squares in the middle, with candidates of 23, 23 and 234. These form a triple.

234 is the full, three-candidate list, and 23 is a subset of it (i.e. all its numbers appear in the full list). Because there are three squares, and none of them have any candidate numbers outside of those in the three-candidate list, they must use up the three candidate numbers (2, 3 and 4) between them.

This lets us remove the 4 from the other two candidate lists in this row, solving their squares.

It's worth looking hard for subset triples. In this example, the 23 lists make an obvious pair (see above), but it's the triple that instantly solves the two outside squares (once you've dispensed with them, you can treat the 23s as a pair again, and use them to solve the 234!). A subset (or 'hidden') triple is often the pattern that will unlock a seemingly impossible puzzle.

Using these rules my Sudoku code can solve all most Sudoku.

Algorithm:

first check the candidate count if it is 3, then go to next cell in the box whoose candidate count is three then check the candidates of them if they are same remove them from all other candidate list.

References: www.paulspages.co.uk/sudoku/