Methods for Solving Sudoku Puzzles

CSCI - 5454, CU Boulder

Anshul Kanakia `kanakia@colorado.edu`

John Klingner `john.klingner@colorado.edu`

1 Introduction

1.1 A Brief History

This number puzzle is relatively new, thought to have first been designed by Howard Garns, a retired architect and freelance puzzle constructor, in 1979.[1] The puzzle was first published in New York by the specialist puzzle publisher Dell Magazines in its magazine Dell Pencil Puzzles and Word Games, under the title Number Place (which we can only assume Garns named). The puzzle was introduced in Japan by Nikoli in the paper Monthly Nikolist in April 1984 as *Suuji wa dokushin ni kagiru* , which can be translated as "the numbers must be single" (literally means "single; celibate; unmarried"). The puzzle was named by Kaji Maki, the president of Nikoli. At a later date, the name was abbreviated to Sudoku.

Sudoku made it's first official digital appearance in 1989. Loadstar/Softdisk Publishing published "DigitHunt" on the Commodore 64, which was apparently the first home computer version of Sudoku. At least one publisher still uses that title.

It has since been re-introduced into popular culture by numerous popular newspapers and magazines such as the New York Post, USA Today, The Boston Globe, Washington Post, and San Francisco Chronicle, not to mention thousands of sudoku apps for computers and mobile devices in recent years.

1.2 The Rules of Sudoku

Sudoku is a number puzzle traditionally comprising of a 9 x 9 square grid of cells where each cell contains a single integer between 1 and 9 (inclusive). The grid is generally further divided into boxes of 3 x 3 cells. The aim of the game is to fill the entire grid up so that a given number appears no more than once in any row, column or box. We are interested in the more general sudoku problem, where we are supplied with a grid of n x n cells. Each box is now a sub-grid of $\sqrt{n}$ x $\sqrt{n}$ cells and each cell may contain the values 1 to n. The rules of the game still remain the same. For the rest of the discussion, we will refer to these rules as constraints. Formally, given an n x n sudoku grid S, we define the sudoku constraints as,

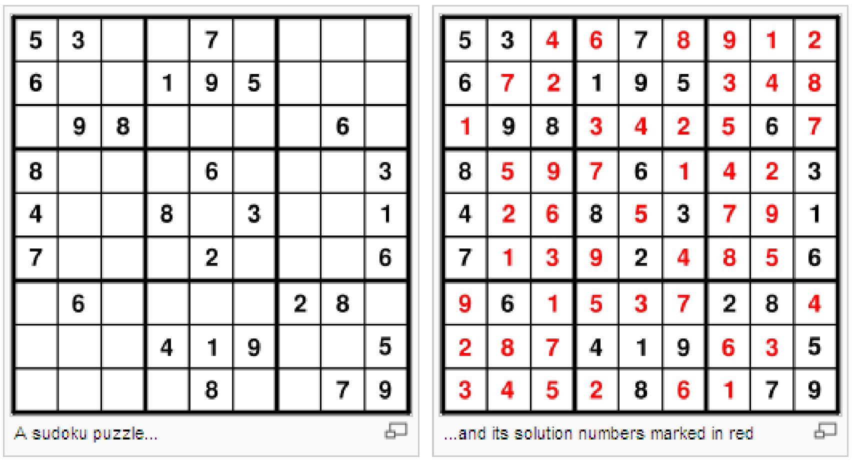
1. **Cell Constraint :** All cells $S_{ij} \in S$ may contain no more than 1 value AND The value must be between 1 and n.

[1]History Source : http://www.spiritustemporis.com/sudoku/history.html

2. **Row Constraint :** All values in rows $S_i \in S$ must be unique, i.e. If $\exists S_{ij} = x$ then $\nexists S_{ij'}$ s.t. $S_{ij'} = x$.
3. **Column Constraint :** All values in columns $S_j \in S$ must be unique, i.e. If $\exists S_{ij} = x$ then $\nexists S_{i'j}$ s.t. $S_{i'j} = x$.
4. **Box Constraint :** We define a box as being a $\sqrt{n}$ x $\sqrt{n}$ sub-grid of S s.t. the top left cell of the sub-grid is always S_{ij} where $(i-1)\%\sqrt{n} == 0$ and $(j-1)\%\sqrt{n} == 0$. We denote a box with the top-left cell S_{ij} as B_{ij}.

   The box constraint thus states that all values in box $B_{ij} \in S$ must be unique, i.e. If $\exists S_{i'j'} = x$, $S_{i'j'} \in B_{ij}$ then $\nexists S_{i''j''} \in B_{ij}$ s.t. $S_{i''j''} = x$.

You can imagine that, even with these constraints there are a lot of valid sudoku grids (roughly 6.5×10^{21} (6.67 sextillion), though only around 5.5×10^{9} (5.5 billion), for the $n = 9$ case. [1]). And so in an attempt to make each sudoku puzzle have only one valid solution they are always supplied with some of the cells already filled in. These cells, often called "clues", contain fixed numbers that cannot be altered by the player while they attempt to fill in the remaining empty cells adhering to the constraints listed above. A Puzzle can be restricted to a unique solution with no fewer than 17 clues, but it is possible to have up to 77 clues and still not have a unique solution.[2] This makes Sudoku puzzle generation an interesting topic of discussion just by itself, which we unfortunately do not discuss here but is definitely worth looking into if you're interested.



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

A sudoku puzzle...

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

...and its solution numbers marked in red

Figure 1: A starting Sudoku with clues in black is shown on the left and it's unique solution is shown on the right.
Image Source - `http://www.comp.nus.edu.sg/ cs1101x/3_ca/labs/07s1/lab7/img/`

The rest of the discussion will concentrate on one of algorithms we can use to find solutions to a given sudoku puzzle. As computer scientists we are interested in two things.

1. Algorithms that can *guarantee* solutions to sudoku problems or,
2. algorithms that run quickly and get you close.

We make this separation in approaches clear because sudoku is a "difficult" problem to solve. All currently known algorithms that *guarantee* a solution to a unique n x n sudoku grid with clues are worse than polynomial time algorithms. Note that if we are given a completed sudoku puzzle we can verify it's correctness in polynomial time but so far no polynomial time algorithm exists for guaranteeing a solution to a given sudoku puzzle. This puts it in a class of problems known as NP-complete problems (Computational complexity is covered in detail in one of the future CSLs). We will concentrate on a specific algorithm of that first kind, i.e. One that guarantees us a solution but runs in exponential time.

2 Backtracking

1. This is a brute force approach that searches through the full space of possible grids. [3]
2. The input to this algorithm is a partially completed Sudoku grid.
   (a) We assume that the partially completed input is valid, or has exactly one solution.
3. The output is a fully completed Sudoku grid that hasn't
   (a) Violated any sudoku constraints, or
   (b) Changed any of the initially completed values.
4. Think of Backtracking as a depth-first search through a "solution tree". In this solution tree:
   (a) Each internal node is a partially completed sudoku grid.
   (b) Each leaf is a completed sudoku grid.
   (c) In this tree, there exist "solution paths", or sequences of choices of children that lead to to a correct, solved grid.
   (d) Say some node has m empty cells in its associated sudoku grid. Then, it will have $n * m$ distinct children, since each empty cell could have n possible values.
      i. It's helpful to think of these $n * m$ children in "clusters" (not in the graph theory sense, just conceptually). There are m clusters, each consisting of n children.

A. If we are on a solution path, then exactly one of the n children in each cluster is also on the solution path. This is equivalent to my saying that for each cell in the Sudoku grid, there is exactly one correct digit I can write.

ii. Depth of the tree from this point is m.

iii. Easy to see how large the problem space is, though there are redundant nodes.

5. When we get to a leaf, we terminate our DFS if it is a valid solution, and return.

6. Now, backtracking is a little bit smarter than just a DFS like I've demonstrated. In backtracking:

   (a) At each node, before looking at its children, we check to make sure the populated grid cells comply with the Sudoku constraints. This way, we only continue down a search path until we're sure it won't lead us to a solution. This can significantly reduce the amount of time spent traversing the tree.

   (b) Every time we try writing the right number down in a cell, or make a choice on a solution path, we've effectively reduced our problem size.

      i. We could have gotten to this node by a partially completed DFS, or we could have just started with this grid as our clues; all the same to the algorithm.

      ii. If we're on the solution path, we'll never go back up.

7. Here's the psuedo-code which works recursively, from the starting Sudoku grid of clues.

```
function BACKTRACK($\mathcal{S}$)
    if $reject(\mathcal{S})$ then
        return null                                   ▷ Prune the rest of this subtree.
    else if $accept(\mathcal{S})$ then
        return $\mathcal{S}$                                         ▷ We're done!
    end if
    $childQueue \leftarrow getChildQueue(\mathcal{S})$
    $\mathcal{S}' \leftarrow childQueue.next(\mathcal{S})$
    while $\mathcal{S}' \neq$ null $\wedge$ $solution ==$ null do
        $solution \leftarrow backtrack(\mathcal{S}')$
        $\mathcal{S}' \leftarrow childQueue.next(\mathcal{S})$
    end while
    return $solution$
end function
```

   (a) The first thing we do when given a new node, ie. a partially completed solution grid ($\mathcal{S}$, is insure that the numbers filled out so far don't violate any of the

Sudoku constraints. If they do, we return null and don't search any of this nodes children. This is the *reject* function.

(b) Next, we check whether $\mathcal{S}$ is a complete and valid grid. If it is, we're done, and we return it. This is the *accept* function.

(c) Finally, we loop through each of $\mathcal{S}$'s children, recursively calling *backtrack* on each of them. This part of the code is just a recursive DFS. Until either:

   i. We found a solution, are done, and return it.

   ii. We've tried all of our children, in which case we return **null**.

(d) Second half is DFS, adding the first half makes it backtracking.

(e) The last function to talk about is *getChildQueue*($\mathcal{S}$). At its simplest, if $\mathcal{S}$ has m cells that still aren't filled out, *getChildQueue* can return the full set of n distinct values for each of the m distinct cells in arbitrary order.

   i. The algorithm will terminate with the correct solution, since our DFS exhaustively searches the space of valid solutions.

   ii. This algorithm can be made smarter, however. Ideally, the *childQueue* will have solutions on the path to the final solution first. If we're on the wrong path, *childQueue* would give use solutions leading to a violation quickly. (Better to find the contradiction in a few steps than to fill out nearly the whole grid before realizing we made a bad choice long ago.)

   iii. Anshul will be discussing these more in a moment, but first a few notes to close this section:

8. Note that, while we have been talking about backtracking as it applies to $n \times n$ Sudoku puzzles, this approach could be useful for any problem that can be thought of in this "solution tree" way, with incremental partial solution steps on the way to a full solution.

9. Backtracking vs. Other Approaches

   (a) Backtracking is guaranteed to find the optimal solution in finite time. This is not the case for all other approaches to these Hard problems, like the simulated annealing technique we implemented in Problem Set 4.

   (b) If your initial Sudoku grid is close to a final solution (ie., you have an easy Sudoku puzzle), backtracking tends to run more quickly than other methods, though it's slower on average.

3 Smarter Backtracking

From the basic backtracking algorithm described so far, it is clear that our run time depends on how many possibilities the `GetChildQueue` function returns. Each of the values returned by this function correspond to a different path along our search tree. So naturally, we'd like to make this function "smarter" by,

1. reducing the number of possibilities returned, which effectively prunes our search tree, and

2. ordering the possibilities returned so we take ones that could get us to the solution faster or help us find a conflict faster.

Algorithm 1

```
function GetChildQueue(S)
    $minQueue$                                ▷ This is a min-heap ordered by cellSet.length
    for $i \leftarrow 1 \dots n$ do
        for $j \leftarrow 1 \dots n$ do
            if S[i, j] == NULL then
                new $cellSet$
                $cellSet.i \leftarrow i$
                $cellSet.j \leftarrow j$
                $cellSet.set \leftarrow Possibilities(S, i, j)$
                $minQueue.push(cellSet)$
            end if
        end for
    end for
    return $minQueue$
end function
```

The `Possibilities` function is where most of the magic happens. This function can be made as simple or as complicated as we'd like. There are two logical extremes here. `Possibilities` could just return all the values 1 through n for that cell without checking anything and would take $O(n)$ time to run. On the other hand, `Possiblities` could essentially eliminate the entire search tree for the particular cell passed to it and return just it's correct value. This is equivalent to solving the sudoku problem where all filled cells are clues and hence takes exponential time. We want something in the middle here. Fortunately, we can use many of the rules humans use to solve sudoku puzzles and implement some of them in `Possibilities`.

- **Naked Single :** The means that cell S_{ij} can have only one candidate number.

Algorithm 2

```
function POSSIBILITIES(S, r, c)
    $retList \leftarrow newList\{1, \ldots, S.length\}$
    for $i \leftarrow 1 \ldots S.length$ do
        if $S_{ri}$ != NULL then
            retList.remove($S_{ri}$)
        else if $S_{ic}$ != NULL then
            retList.remove($S_{ic}$)
        end if
    end for
    for each cell $s$ in $S.getBox(r, c)$ do
        if $s$ != NULL then
            retList.remove($s$)
        end if
    end for
    return $retList$
end function
```

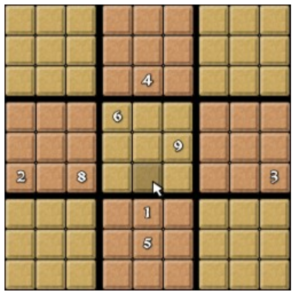


Figure 2

- **Hidden Single :** If a region contains only one square which can hold a specific number then that number must go into that square. This is considerably more complicated to implement in code even though it happens to be one of the easier tricks to solving sudoku by humans.[2]

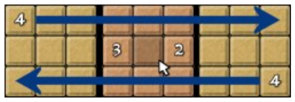


Figure 3

[2]Image Credits : `http://www.su-doku.net/tech.php`

So it is easy to see now how implementing `Possibilites` with these tricks could reduce the length of list returned, which effectively prunes our search space.

4 Why is the Min-Heap Ordered that Way

1. In closing, we wanted to spend a bit more time trying to motivate this idea that the *minQueue* should be ordered by the size of the cell sets.
2. Recall the possibilities function, and lets say you implemented your *possibilities* function to just use the first (naked single) "trick" described, but your *minQueue* has arbitrary order, you won't have helped yourself very much.
   (a) All of the child nodes pruned by applying the naked single trick are those nodes which would have immediately resulted in a $reject(\mathcal{R})$ returning **true**. But implementing this min-heap and doing this work in the possibilities function helps a lot because it lets us order cell sets by the number of possibilities. Why?
3. A bit more formally (but not as rigorous as we would have liked).
   (a) As I said earlier, making good choices, ie. choices on the solution path, reduces the size of our problem by:
      i. Reducing the remaining depth of our tree, and
      ii. Reducing the number of possibilities for other cells.
      iii. So we want to do this as often as possible.
   (b) If there are ever cells with exactly one possibility, then its clear that those cells should be prioritised over others.
   (c) More generally,
      i. If the possibilities function has told us that one cell set (or cluster, after pruning) has two elements (that's two possible values that don't violate constraints), but another cell has three elements, and we pick elements of the cell set arbitrarily, then we have a 50% chance of staying on the solution path with the first cell, but only a 33% chance with the second cell.
   (d) If we're on the wrong path, note that exhausting a cluster is equivalent to trying all possibilities for a cell, and thus means we should return. The smaller a cluster is, the easier it is to exhaustively check.

5 Advanced Sudoku Solving Algorithms

As we've seen so far, efficiently solving an n x n grid Sudoku is a very interesting problem to think about and this brings us to the seconds broad class of algorithms that attempt to do this. Though we may not have a chance to discuss any of these we would like to quickly highlight some of our favorites! Many of the following algorithms attempt to convert Sudoku to a "Constraint Optimization" problem. These are the sorts of problems we looked at when talking about simulated annealing earlier in the semester and indeed, some of the algorithms described below use simulated annealing to attempt to solve sudoku grids.

1. **Solving sudoku puzzles using artificial bee colonies**[4] : As both of us work in the Swarm Robotics lab on campus we felt it was apt to at least mention one Swarm Algorithm for solving sudoku. This paper describes the use of specialized artificial bee populations used to explore the search space of the sudoku grid.

2. **The Chaos with sudoku**[5] : This paper first outlines a general method for converting a sudoku grid to another well known NP-Complete problem called "Boolean Satisfiability". This process of converting one NP-complete problem into another well known problem in NP is called "Reduction" and is a common tool used by computer science theorists to prove that a problem is NP-complete. This paper also goes on to describe and analyze how "difficult" a given sudoku grid (with clues) is to solve.

3. **Sudoku as a Constraint Problem**[6] : This paper attempts to rephrase the sudoku puzzle as a constraint problem and discusses implementations for solving the problem using techniques such as flow algorithms and bipartite matching. This is a good introductory paper to solving sudoku as a constraint problem and constraint problems in general.

4. **The Boltzmann Machine**[7] : This undergraduate dissertation is an excellent overview of some of the popular algorithms used to solve sudoku including backtracking and it's variants. In fact, as a short acknowledgment, we would like to thank the authors of this article as it helped us immensely when preparing examples and explanations for our talk.
   One of the algorithms mentioned in this paper that we did not discuss is the "Boltzmann Machine" algorithm. This algorithm begins by converting the sudoku grid of integers into a 3 dimensional grid of booleans. It then further converts this 3D grid of booleans into a large fully connected graph with very special properties and operations. This class of graphs is often called an "Artificial Neural Network" (AAN) and very specifically, the paritcalar AAN we use to solve the sudoku puzzle is called a Boltzmann Machine.

We hope you enjoyed this brief introduction in the world of mathematics and computer science as seen through the ideas of sudoku solvers! Feel free to email us if you have any questions about the topics we covered in our talk or ones mentioned in these notes.

References

[1] E. Russell and F. Jarvis, "There are 5472730538 essentially different sudoku grids ... and the sudoku symmetry group." `http://www.afjarvis.staff.shef.ac.uk/sudoku/sudgroup.html`, September 2005.

[2] G. McGuire, B. Tugemann, and G. Civario, "There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem," *arXiv preprint arXiv:1201.0749*, 2012.

[3] E. Gurari, "Backtracking algorithms." `http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch19.html#QQ1-51-128`.

[4] J. A. Pacurib, G. M. M. Seno, and J. P. T. Yusiong, "Solving sudoku puzzles using improved artificial bee colony algorithm," in *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on*, pp. 885–888, IEEE, 2009.

[5] M. Ercsey-Ravasz and Z. Toroczkai, "The chaos within sudoku," *Scientific reports*, vol. 2, 2012.

[6] H. Simonis, "Sudoku as a constraint problem," in *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, pp. 13–27, Citeseer, 2005.

[7] P. BERGGREN and D. NILSSON, "A study of sudoku solving algorithms,"