Executive Summary: Sudoku Algorithm Comparison

Introduction: Sudoku is a logic-based number placement puzzle. The objective is to fill every square in a 9x9 grid with a number between 1 and 9 so that each row, column and sub 3x3 square has exactly one of each number.

Algorithms discussed: The algorithms that were chosen are: Depth First Search, Simulated Annealing, and Dancing Links.

Goal: To solve a 9x9 Sudoku puzzle in a reasonable amount of time using a reasonable amount of memory and processing power.

Results:

Algorithm	Complete?	Memory	Average Runtime	Worst Runtime
Depth First Search	Yes	567 bytes	361 days	1.9821 years
Simulated Annealing	Not always	15.45 megabytes	5.87 seconds	15 seconds
Dancing Links	Yes	0.9023 megabytes	1435 milliseconds	2385 milliseconds

Analysis and Conclusion: Both Simulated Annealing and Dancing Links are very viable solutions. Simulated annealing will find the solution if one exists almost 100% of the time for 9x9 puzzles and does so with relatively low memory overhead and runtime. Depth first search although it is complete and finishes in a set amount of time, albeit a very long amount of time, is not very useful for this problem. By the time you get an answer back you probably won't even like Sudoku anymore.

Simply because Dancing Links runtime is much more consistent than Simulated Annealing as well as being guaranteed complete makes it a much better choice. **Therefore it is my suggestion to use Dancing Links to solve Sudoku Puzzles.**

Sudoku Algorithm Comparison

Kent Sommer
University of Minnesota
Introduction to Artificial Intelligence 4511W

Contents

1 The Puzzle

Sudoku is a logic-based number placement puzzle. The objective is to fill every square in a 9x9 grid with a number between 1 and 9 inclusive so that each row, column and sub 3x3 square has exactly one of each number. Solution numbers are marked in red for this 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

2 Constraints and properties

Hardware and data set constraints:

- Macbook Air with Intel i5 dual core processor clocked at 1.8GHz
- 8GB of RAM.
- Integrated GPU.
- All sets of puzzles fed to solvers are identical

Properties:

- Inconsistent branching factor
- Large state space

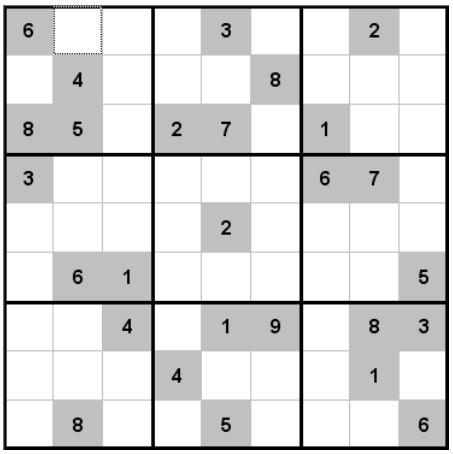
3 Choice of Algorithms

In order to determine the best algorithm for Sudoku, I have chosen to look at Depth first search, Simulated Annealing, and Dancing Links. Both Depth first search and Dancing Links are brute force algorithms to some extent while Simulated Annealing is not. Each algorithm will be assessed by memory usage, speed, and completeness.

4 Depth First Search

4.1 Description and application

The following description refers to this puzzle:



Depth first search for Sudoku will start at the first empty square on the board (in this case row 0, column 1). Taking into account the constraints of Sudoku, the only available numbers for the square are 1, 7, or 9. This leaves depth first search with three branches to look at. Since depth first search goes down the left side of a tree first, 1 is selected. Depth first search then goes to the next option (row 0, column 2) and has the option of either 7 or 9. 7 is selected as it is the leftmost branch and we haven't had to backtrack yet. The next node is selected (row 0, column 3) and has the children 5 and 9. Depth first search will continue like this until one of two things happen. Either every square on the board has been filled and thus the puzzle has been solved, or it gets to a square that has no options in which case it backtracks to the previous square and tries the next branch. If depth first search has looked at every possible assignment with the first square being 1 and has not found a solution it will backtrack to the first square and assign it the next left child which is 7 before repeating the above process.

4.2 Completeness

Depth first search will examine every possible branch until a solution is found or until it reaches the end of the tree at which point the puzzle can be declared unsolvable. Because of this, depth first search is complete and if a solution exists it will find it.

4.3 Memory

Memory usage is where depth first search does best. For a 9x9 Sudoku puzzle, all that needs to be stored on the host machine is the array for the puzzle. I described a decision tree above, but it isn't a needed data structure, it is simply an easy way to visualize the problem. To determine what to do next, all that is required is the current state. The array for the puzzle needs two things: an integer, and a Boolean to mark each entry as valid or invalid. This means total memory will not exceed 567 bytes, which can easily be stored in RAM.

$$((9 * 9) * 4) + ((9 * 9) * 1) = 567_{bytes} \tag{4.1}$$

4.4 Run time

Run time will depend on a number of different properties, all of which effect how many nodes depth first search will have to inspect. The first thing to note is that depth first search will take longer or shorter depending on what number the first square it tries to fill in needs to be. For instance, in the walk through above it started at 1, but what if the first square to be filled in actually needed to be 7? Depth first search would need to look at all the possibilities on branches 1 through 6 before finally reaching the solution branch.

If we assume that the given Sudoku puzzle only has 17 pre-assigned numbers (the minimum to ensure only one solution) then the depth of the tree can be calculated.

$$(9 * 9) - 17 = 64 \tag{4.2}$$

Branching factor is not consistent and thus cannot be given a hard number. To estimate branching factor we will assume that all nodes take the average branching factor calculated by combining the minimum and maximum assignments. The maximum assignment options for an arbitrary square are 7 (this happens when there is only one number assigned in its row column and sub space and all are the same number). The minimum

assignment is 1 (when whole row, column, or sub space is completely assigned save for one square). Therefore the average branching factor will be 4.

State space can then be calculated using this average branching factor and given depth.

$$4^{64} = 3.4028366 * 10^{38} \tag{4.3}$$

For most Sudoku puzzles this will be an over estimation, however, it is not orders of magnitude off and thus error is not significant. For each node, depth first search will do two things. First it will check the current number against its row, column, and sub space and make sure it satisfies the all-different constraint (8 operations done 3 times). Second depending on whether or not the given node fails or passes the all-different check depth first search will move to the next node or backtrack to the previous and pick its next branch (1 operation done 1 time). This gives a total of (8 * 3) + 1 = 25 total operations per node.

$$(3.4028366 * 10^{38}) * 25 = 8.5070592 * 10^{39}\ operations \tag{4.4}$$

Let us assume that this search is running on a 3.4GHz quad core processor. This gives us a total search time worst case of:

$$\frac{8.5070592 * 10^{39}}{(3.4 * 4) * 10^{9}} = 1.9821\ years \tag{4.5}$$

Best case the solution is down the left most branch of the tree and depth first search only has to look at 1/9th of the tree, this still takes 80 days. Although the memory consumption is very low, it is impractical to use depth first search to solve most Sudoku puzzles due to the extremely long time it takes. On average, the solution will fall somewhere in the middle of the tree meaning it would take 361 days to run.

Algorithm	**Complete?**	**Memory**	**Average Runtime**	**Worst Runtime**
Depth First Search	Yes	567 bytes	361 days	1.9821 years

5 Simulated Annealing

5.1 Description and application

Simulated annealing works by solving a global fitness value. For simulated annealing in Sudoku, this means every square is filled and every rule or constraint is satisfied for each square. Once the number of squares that break the constraints reaches zero (fitness

value of 0), the puzzle is solved. For Sudoku, simulated annealing starts by filling all empty squares with randomized values between 1 and 9 inclusive. A fitness value is applied to each square as well as the board as a whole. Squares are swapped and the fitness functions for both squares and the board are re-evaluated. If this swap improves on the fitness score it is kept as the new starting board for future swaps. This process would be equivalent to the cooling in simulated annealing. In order to avoid getting stuck on local maxima (board that has an overall close to zero fitness function but is almost totally incorrect), it will save a few "worse" boards depending on the temperature. This is equivalent to the heating in simulated annealing. Without it, this algorithm could very easily get stuck trying to improve on an unsolvable board.

5.2 Completeness

Simulated annealing is not necessarily complete, but can be at the cost of taking longer. To determine completeness it is important to first understand what simulated annealing takes as parameters (initial temperature, cooling rate, and lowest temperature needed before reheat). If the initial temperature is too high, the algorithm will randomly swap squares for a longer time. If the initial temperature is too low, it can keep the algorithm from getting out of local maxima. An initial temperature around 40 is probably best as it still allows a sufficient amount of swaps that increase the fitness cost (75-80%) but doesn't increase arbitrary swaps by too much. If the temperature before reheat is too high, the algorithm will never "cool down" enough to find a solution. If the temperature before reheat is too low, there is an increase in time spent in local maxima before reheating. Therefor, it is best to select a safe guess on the high side for initial temperature and a safe guess on the low side for temperature before reheat. If safe guesses are taken and a slow cooling rate is used then simulated annealing will most likely be complete but it is not necessarily complete due to its unfortunate ability to get stuck on local maxima.

5.3 Memory

Simulated annealing will require a few arrays to function. It needs a 9x9 current state two-dimensional array that holds the current assigned number and the fitness value in each slot. It needs a temporary 9x9 two-dimensional array to hold the potential swap and calculate the fitness function. There is also some number of two-dimensional arrays saved that can be returned to if there is a need to reheat. The algorithm will also have two integer variables to store the current global fitness cost and the swapped global fitness cost so that it can compare the two and determine if it should take the new board. Using the above information as well as each slot in the array holding 2 integers (8 bytes), minimum and maximum memory cost can be calculated. Minimum usage would happen if you don't keep any boards for reheating and only used the two arrays ((9*9) * 8 +

(9*9) * 8 = 1296 bytes). Maximum memory usage is hard to estimate. Starting from the initial temperature you could potentially accept all bad swaps, however, as time went on you would start accepting less and less till eventually you would only accept better swaps. Let us assume that to start, we accept 100% of swaps. During the first section of time we have swapped 10,000 elements leaving us with 10,000 boards to store. Since the algorithm has cooled we only keep 7,500 new boards. This continues until we don't accept any bad swaps or new boards leaving us with a total of:

$$10,000 + 7,500 + 5,000 + 2,500 + 0 = 25,000\ boards \tag{5.1}$$

Maximum memory based on the above example is then:

$$((9 * 9) * 8) * 25,000 + ((9 * 9) * 8) * 2) = 15.45\ megabytes \tag{5.2}$$

5.4 Run Time

Since run time can vary so much based on initial temperature, cooling rate, etc. I tested run times using the Simulated Annealing Solver by Ricardo Godoy De Oliveira [1]. Over a set of 100 puzzles (rated hard) I had results in the range of 2.4 - 15 seconds. The average run time was 5.87 seconds. This is a huge step better than the brute force approach that depth first search uses. I would also like to note that memory usage stayed under 12 megabytes for each puzzle so average memory usage real world is minimal and will still fit in RAM just like depth first search.

Algorithm	**Complete?**	**Memory**	**Average Runtime**	**Worst Runtime**
Simulated Annealing	Not always	15.45 megabytes	5.87 seconds	15 seconds

6 Dancing Links

6.1 Description and application

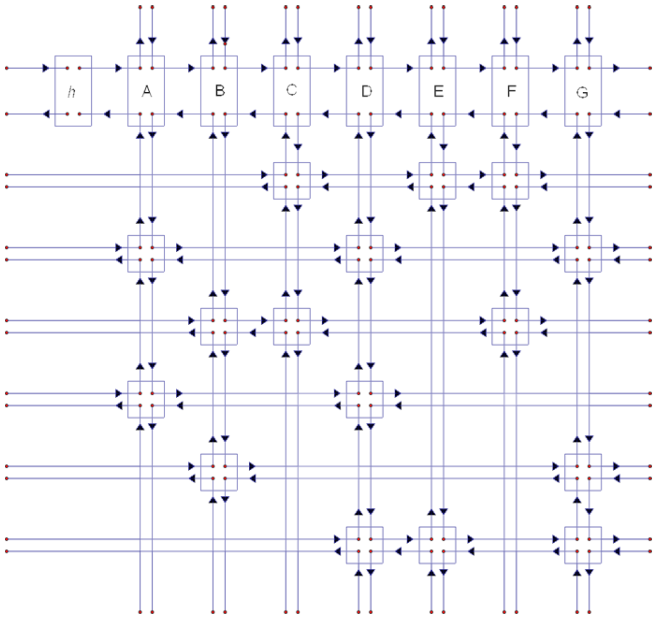
Dancing Links takes the basic idea that removing an element and inserting it back into a linked list runs in constant time, hence the name. Dancing Links is actually an algorithm designed for Exact Cover problems so its usable applications are somewhat limited, however, Sudoku can be presented as an exact cover problem!

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{6.1}$$

Given the matrix above, Dancing Links will try to find some set of rows in which there is just one 1 in each column. For this matrix that would be rows 1, 4, and 5. All that needs to happen to allow Dancing Links to solve Sudoku is to represent the puzzle as an exact cover problem like the matrix above.

Dancing Links would represent Matrix (4.1) as a circular doubly-linked list. Every column has a special column node or head which contains that column's name and size as well as the number of nodes in the column. Only entries in the matrix that are 1's are represented as nodes. Each node points to five different things: The node above, below, to the left, to the right, and the column node or head node. The illustration below shows how Dancing Links would represent the Matrix given above in (4.1).

Where Dancing Links gets its large performance boost is the cover and uncover functions. Cover removes a column from the linked list structure as well as all the rows in the column from the other columns they are in. This greatly shrinks searchable problem size as the algorithm runs down each possibility and allows it to quickly determine that a choice was incorrect and then to backtrack. Since the removed nodes retain their placement information, you can simply do a reverse of the cover operation.



Linked Representation of Matrix (4.1)

6.2 Completeness

Since Dancing Links is simply an optimization to brute force back tracking, it will always find a solution if one exists.

6.3 Memory

To examine the memory usage of Dancing Links it is important to understand how it works in relation to a Sudoku puzzle. Imagine you were a magician and you have a set of tricks and each trick shows a few different skills you have. To have a successful show you must pick a group of tricks that combined show off all of your skills, but shows each skill only once. For Sudoku, your skills form the columns and the tricks to pick from form the rows. Let us first examine the "skills" or requirements for Sudoku: Every square must have a number in it, every row must have the numbers 1 through 9 inclusive in it, every column must have the numbers 1 through 9 inclusive in it, and every box must have the numbers 1 through 9 inclusive in it. Each of these requirements happens on every square therefore the total number of requirements is 9*9 = 81 * 4 = 324. This means there are 324 columns in the matrix.

Let us now examine the "tricks," or in the case of Sudoku, the options. For any square it is possible to place a number 1 through 9 inclusive which means the total number of options is 9*9*9 = 729. One is then added to account for the header row, which allows each "node" the ability to point to the "head" node of its column for fast access bringing the total number of rows to 730.

Applying this information, each row (number assignment 1-9) solves a set of requirements. For example, assigning 5 to square (0,0) solves the requirements: A number must be in (0,0), row 0 must have a 5, column 0 must have a 5, and box 0 must have a 5. At most, the memory needed will be what is needed to store the linked list implementation of the given sparse 730x324 matrix.

$$(730 * 324) * 4 = 0.9023\ megabytes \tag{6.2}$$

6.4 Run Time

Dancing Links as mentioned earlier is simply a brute force algorithm with backtracking on a circular doubly linked list. A solution is found once all the columns have been

removed from the matrix. In order to achieve this, every row that has been added to the answer has one node in every column. Once you remove or "cover" a node everything you need to re-insert the node into the linked list to backtrack is already there. All removing and re-inserting can be done in place and efficiently due to the low cost of pointer manipulation.

Running a C implementation written by Xi Chen [2] gave a solution speed of 1435 milliseconds on average. The reason it is so low is that back tracking costs almost nothing (simply re-insert the last removed node and start from there) and because as you start removing nodes you process in larger and larger chunks. This means you realize you have a bad choice earlier than most other brute force searches (plain depth first search for example). Since it is a brute force search, problem difficulty assignment won't effect run time.

Algorithm	**Complete?**	**Memory**	**Average Runtime**	**Worst Runtime**
Dancing Links	Yes	0.9023 megabytes	1435 milliseconds	2385 milliseconds

7 Comparison

7.1 Summary

Algorithm	**Complete?**	**Memory**	**Average Runtime**	**Worst Runtime**
Depth First Search	Yes	567 bytes	361 days	1.9821 years
Simulated Annealing	Not always	15.45 megabytes	5.87 seconds	15 seconds
Dancing Links	Yes	0.9023 megabytes	1435 milliseconds	2385 milliseconds

7.2 Choice

Both Simulated Annealing and Dancing Links are very viable solutions. Simulated annealing will find the solution if one exists almost 100% of the time for 9x9 puzzles and does so with relatively low memory overhead and runtime. Depth first search although it is complete and finishes in a set amount of time, albeit a very long amount of time, is not very useful for this problem. By the time you get an answer back you probably won't even like Sudoku anymore. Because Dancing Links runtime is much more consistent than Simulated Annealing and it is guaranteed to find a complete solution, it is a much better choice. **Therefore, it is my suggestion to use Dancing Links to solve Sudoku Puzzles.**

References

[1] https://github.com/rgoliveira/sudoku-sa-solver

[2] https://github.com/kybernetikos/dancing-links