Zendoku puzzle generation

Gareth Rees, 2007-06-10

1. Introduction

This article describes some of the technical challenges involved in implementing a sudoku puzzle generation algorithm on a handheld video game console, and how the developers at Zoonami Limited solved them in the production of *Zendoku*, released in Spring 2007 on the Nintendo DS and Sony PSP.

2. Requirements

Let’s start by considering the requirements the puzzle generator must satisfy.

1. It must generate proper sudoku puzzles: there must be exactly one valid way to complete the grid.
2. It must be able to generate puzzles with a wide range of difficulties, so that we can please players of different skills, and so that we can present a graded series of difficulties in *Zendoku*'s story mode.
3. At each difficulty level, puzzles must not differ very much in difficulty. A player's progress must not depend mainly on how lucky they get with the generator algorithm.
4. It must produce a puzzle quickly on the Nintendo DS. It would be rather unfortunate to have to resort to a "generating puzzle… please wait" screen. Ideally, there should be no more than a second or so between the player selecting a difficulty level and starting to play the puzzle—no more time than it takes to animate the transition from the menu to the game.
5. The method must be cheap to develop, in terms of programmer time, as we had only two programmers with whom to make a cross-platform console game.

3. Strategic overview

The strategy we picked was *generate-and-test*. This uses two algorithms: a *generator* which produces, in an unguided fashion, sudoku puzzles of unknown difficulty; and a *grader*, which works out how difficult each puzzle is. If a puzzle is the wrong difficulty we throw it away and generate another candidate.

This approach may seem rather odd at first sight, even wasteful. It's certainly quite unlike the way that human setters create sudoku puzzles. However, it has two advantages that were key for us: it's straightforward to develop, requiring little in the way of research, and reliable in practice: given a generator that does a good job of sampling the space of puzzles, and a grader that does a decent job of determining the difficulty, getting the puzzle we're after is a simple matter of putting in enough CPU time. Note that this may cause us some difficulties for in creating puzzles fast enough—see **requirement 4**—but we can worry about that later—see **section 6**.

Copying the methods of human setters seemed likely to be impractical in the very limited time available, an exercise suited more for a research project than a commercial game. Also, it seemed likely that to program these methods we would have to become reasonably proficient in them ourselves, and that would also take time we didn't have.

A fallback possibility that we always had in mind was that we could include with the game some pre-prepared puzzles. If our generate-and-test approach proved to be too slow to run on the console itself, we could at least run it in our office and include the output in the game. Or, if we completely failed to produce a decent generator, we could buy in puzzles from a composer. This approach worked for Hudson's ***Sudoku Gridmaster*** which contains "more than 400" built-in puzzles, created by Japanese puzzle company **Nikoli**. However, this would be less suitable for a puzzle fighting game, which ought to be able to support endless replay. We could disguise the fact that we were reusing puzzles from the built-in set by permuting the symbols, rows and columns, but the experience would probably end up being a bit "samey" after a while. It also seemed like a bit of a cop-out!

4. Generator

The basic idea behind our puzzle generator is as follows:

1. Generate a random grid satisfying all the sudoku constraints.
2. Pick a random permutation of the cells in the grid.
3. For each cell in the permuted order: remove the symbol in that cell; check to see if this has left the puzzle with multiple solutions; if so, put the cell back.

This generates an "irreducible puzzle": a grid from which no more cells can be removed without leaving multiple solutions. It's a sudoku puzzle, but at this stage we don't know anything about how hard it is!

For step (3) we need a fast solver (fast because it has to be run many times for each puzzle we generate). And having built a solver, it's easy to modify it to generate the completed grids in step (1), by starting solving with an empty grid and making random choices when available.

The structure of the sudoku puzzle makes it an ideal candidate for representation as a **constraint satisfaction problem**. This approach has two neat aspects: it treats all the constraints equally, so there's no need to code up a distinction between choosing a number to put in a cell and choosing a cell to put a number in; and there's a clever way of implementing the algorithm (Donald Knuth's "Dancing Links"; see **section 4.3**) which allows choices that are ruled out by the constraints to be quickly eliminated, and quickly put in again when backtracking.

4.1 Example: solving a 2×2 Latin square

I'll illustrate the constraint satisfaction approach with a very (very!) small problem: the solution of 2×2 Latin squares. A Latin square is like a sudoku except that there are no blocks, so in this case we have to put the numbers 1 and 2 into the cells of a 2×2 grid so that there is a 1 and a 2 in each row and each column (see **figure 1**).

Figure 1.
Solve this 2×2 Latin square!

You should find it straightforward to check by hand that there are exactly two solutions!

We begin by building a matrix whose columns are the constraints and whose rows are possible choices of placement. In the case of 2×2 Latin squares, there are 12 constraints (3 types of constraint, 2×2 = 4 constraints of each type).

- Constraints 1–4: We must place some number in each cell.
- Constraints 5–8: We must place each number somewhere in each row.
- Constraints 9–12: We must place each number somewhere in each column.

There are 8 choices of placement: either number may be placed in each cell. Expressed as a binary matrix, the combinations of constraints satisfied by each choice is shown in **figure 2**.

Figure 2. Constraint matrix for a 2×2 Latin square.

Choice	Constraint											
	Some number in column				The number				The number			
	1		2		1		2		1		2	
	and row				must appear in row				must appear in column			
	1	2	1	2	1	2	1	2	1	2	1	2
1 at (1,1)	■				■				■			
2 at (1,1)	■						■				■	
1 at (1,2)		■				■			■			
2 at (1,2)		■						■			■	
1 at (2,1)			■		■					■		
2 at (2,1)			■				■					■
1 at (2,2)				■		■				■		
2 at (2,2)				■				■				■

The size of this table should make it clear why I've chosen such a small example: the smallest sudoku, 4×4, has 64 constraints and 64 ways of placing a number, while a 9×9 sudoku has 324 constraints and 729 ways of placing a number. You can see the entire matrix for the latter **illustrated in glorious ASCII** by Robert Hanson.

So how does this constraint matrix help? Well, every solution to the Latin square corresponds to a subset of rows from this matrix such that each constraint is covered exactly once—that is, each column has a single black square in the chosen subset of rows. Computer scientists will recognize this as an instance of the well-known NP-hard **EXACT COVER problem**.

Figure 3 shows the rows corresponding to a solution to the 2×2 Latin square. Note that each column contains a single black square.

Figure 3. A 2×2 solution represented as a subset of rows from the constraint matrix.

1	2
2	1

1 at (1,1)	■				■				■			
2 at (1,2)		■						■			■	
2 at (2,1)			■				■					■
1 at (2,2)				■		■				■		

4.2 “Algorithm X”

We can find solution sets, if any exist, by a straightforward application of depth-first search with backtracking. In particular, we can incrementally build up a set of rows making up a solution as follows:

1. Pick an unsatisfied constraint: that is, a column with no black cell in any of the rows in the solution set. If there are no unsatisfied constraints remaining, the solution set is complete: if all we wanted was any solution, we’re done; if we want all solutions, then make note of the solution we’ve got and then backtrack to the previous time we chose a row and take the next choice instead.
2. Pick a row that satisfies that constraint: that is, one with a black cell in the chosen column. If there is no such row, then we’ve reached a dead end and we must backtrack to the previous time we chose a row and take the next choice instead.
3. Add that row to the solution set.
4. Delete all rows that satisfy any of the constraints satisfied by the chosen row: that is, all rows that have a black cell in the same column as a black cell in the chosen row.
5. Return to step 1.

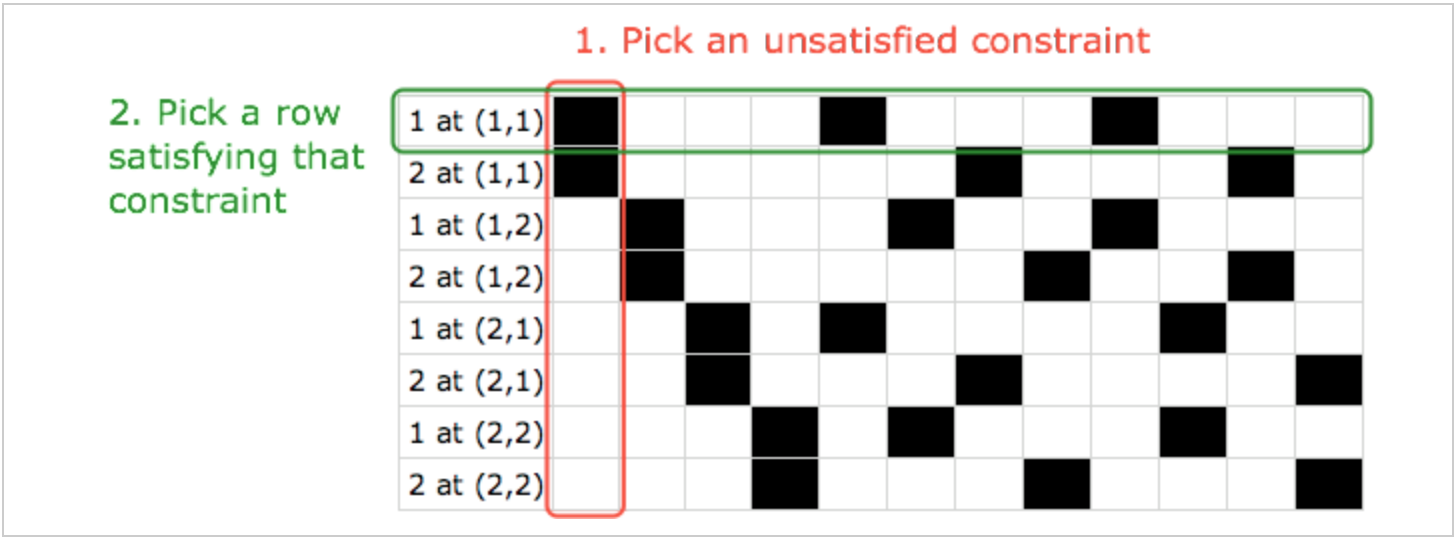
Donald Knuth **comments** on this algorithm, “I will call [it] algorithm X for lack of a better name […] Algorithm X is simply a statement of the obvious trial-and-error approach. (Indeed, I can’t think of any other reasonable way to do the job, in general.)”

The choice of constraint at step 1 doesn’t affect the correctness of the algorithm, but makes a great deal of difference to its running speed. The obvious heuristic is to always pick the constraint satisfied by the smallest number of rows, as this reduces the branching factor of the search. Knuth calls this the “*S* heuristic”; see **section 4.4** for how well it performs in sudoku.

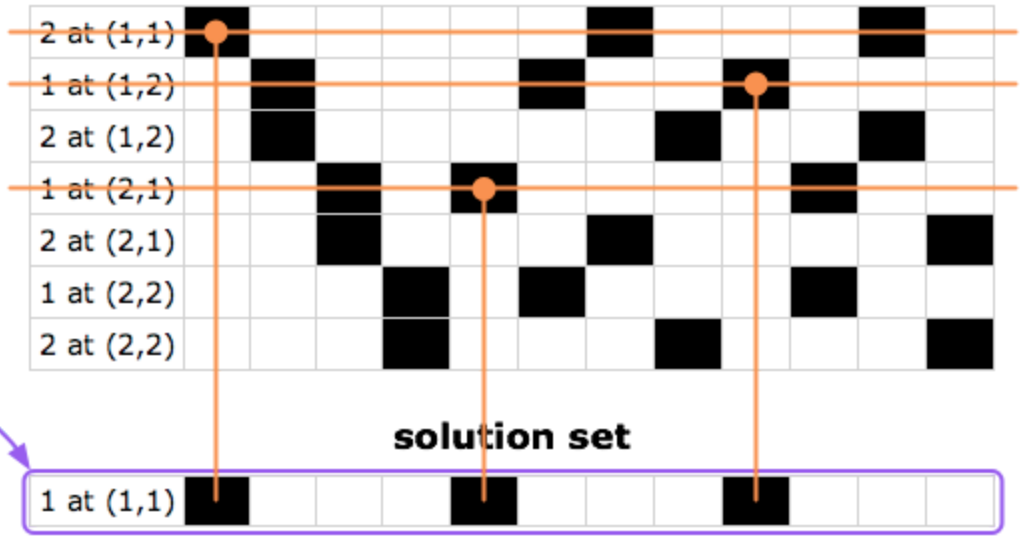
The choice of row at step 2 doesn’t affect the correctness of the algorithm either, so long as the backtracking makes sure to visit all satisfying rows, but if there are multiple solutions the choice does affect the order in which the solutions are generated.

Figure 4 shows this algorithm applied to our example. The algorithm makes a bad decision at step 6 and has to backtrack.

Figure 4. Solving the 2×2 Latin square using Algorithm X.

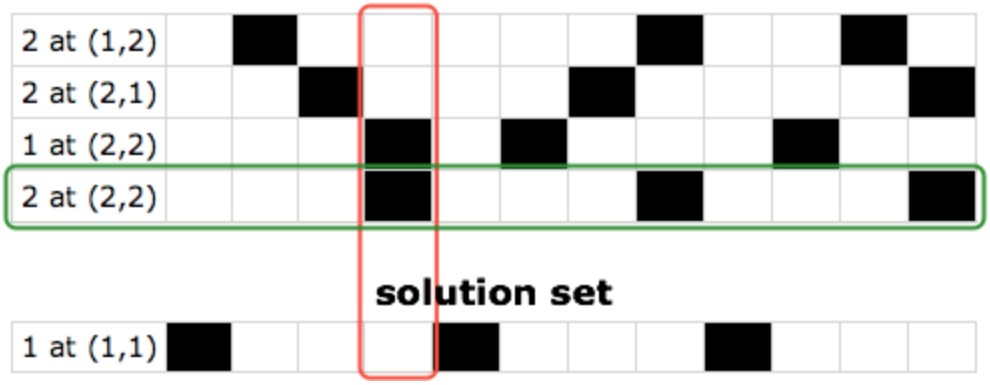


4. Delete all rows that satisfy any of the constraints satisfied by the chosen row



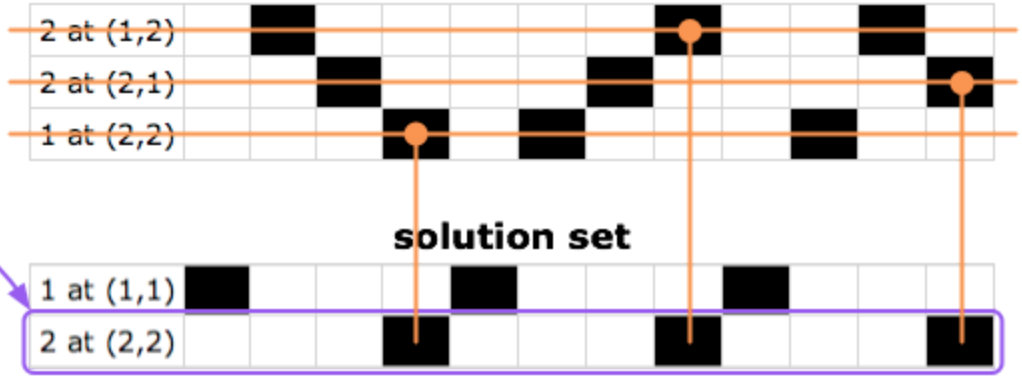
3. Add the row to the solution set

5. Pick an unsatisfied constraint

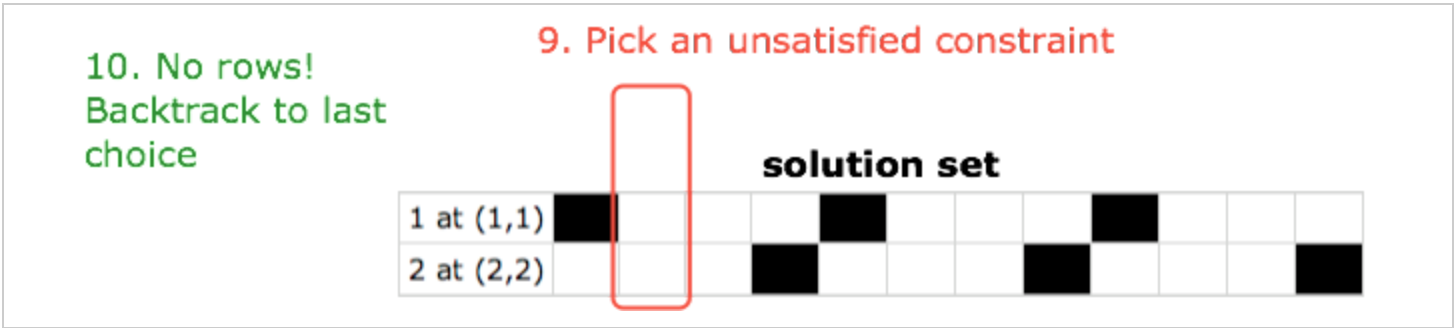


6. Pick a row satisfying that constraint

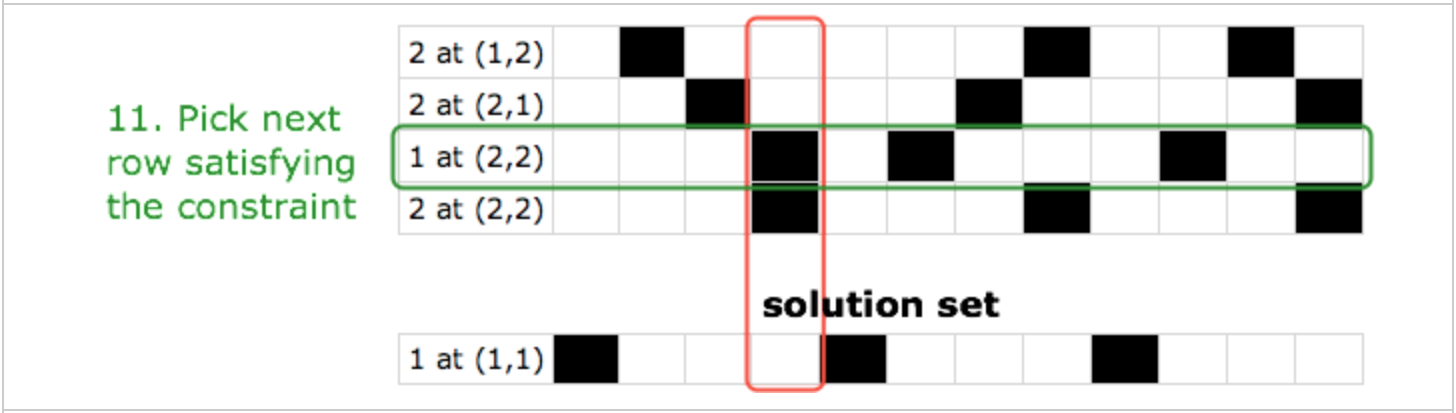
8. Delete all rows that satisfy any of the constraints satisfied by the chosen row

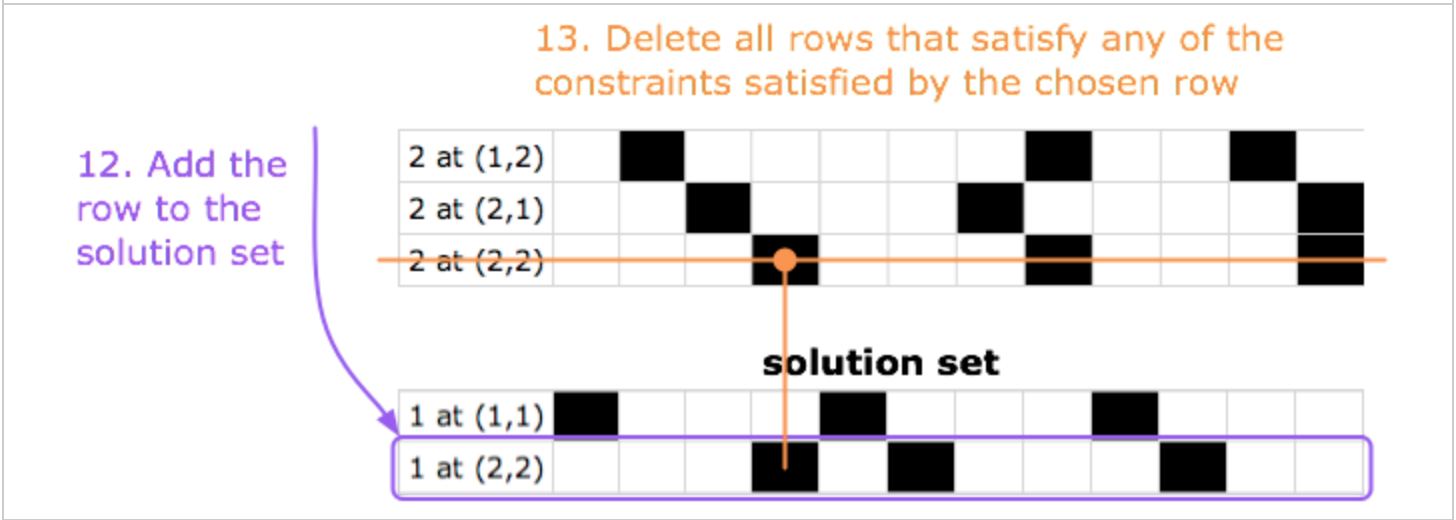


7. Add the row to the solution set



The last choice was at step 6, the selection of a row that satisfies the fourth constraint, so we go back and pick the next row satisfying the constraint.





And so on. The remaining two rows will be added to the solution set one at a time. It doesn't matter which choices are made now, the algorithm can make no more mistakes.

4.3 "Dancing Links"

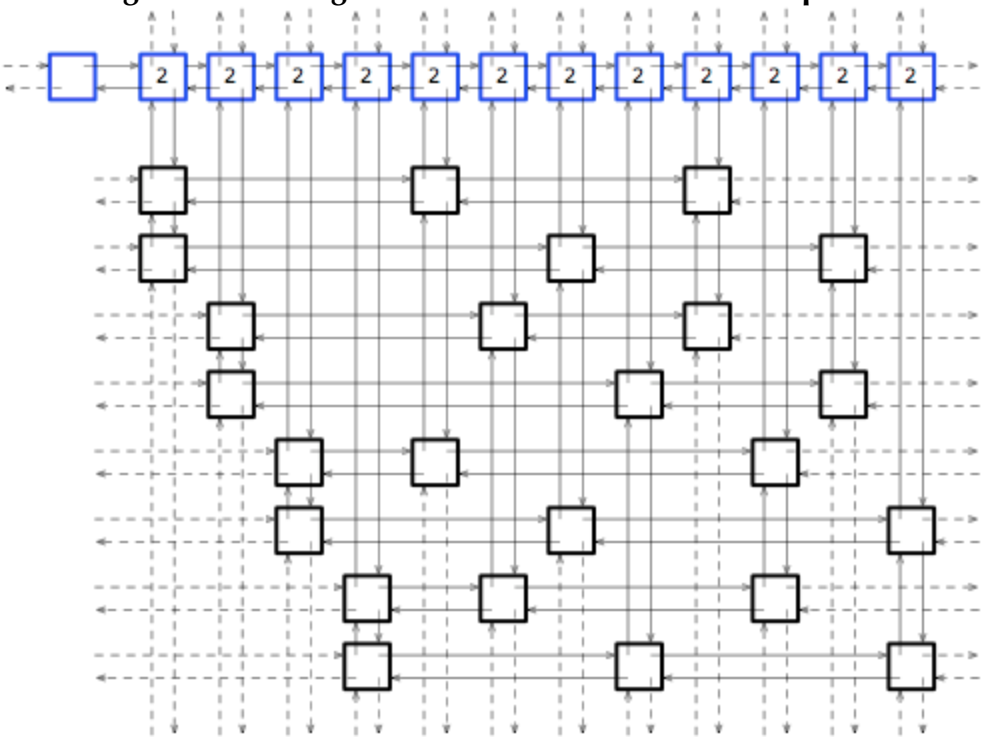
Donald Knuth describes an efficient technique for implementing Algorithm X, which he calls "**Dancing Links**". This starts with the observation that in typical instances of the EXACT COVER problem, the number of constraints satisfied by each row is quite small. This is particularly true in 9×9 sudoku, where the full constraint matrix has 324 columns and 729 rows—236,196 cells in all—but only 2,916 of those cells are occupied.

We need to be able to search efficiently along columns (in steps 2 and 4) and along rows (in step 4), so we'll make linked lists for each row and column. We need to be able to efficiently remove cells from their column (in step 4) so we'll use doubly-linked lists. This means that each occupied cell has four pointers, going to the occupied cells to the left, right, above and below. We also need to be able to pick

the best column to search (at step 1) so we'll have additional data structures representing columns, with a count of the number of occupied cells in each.

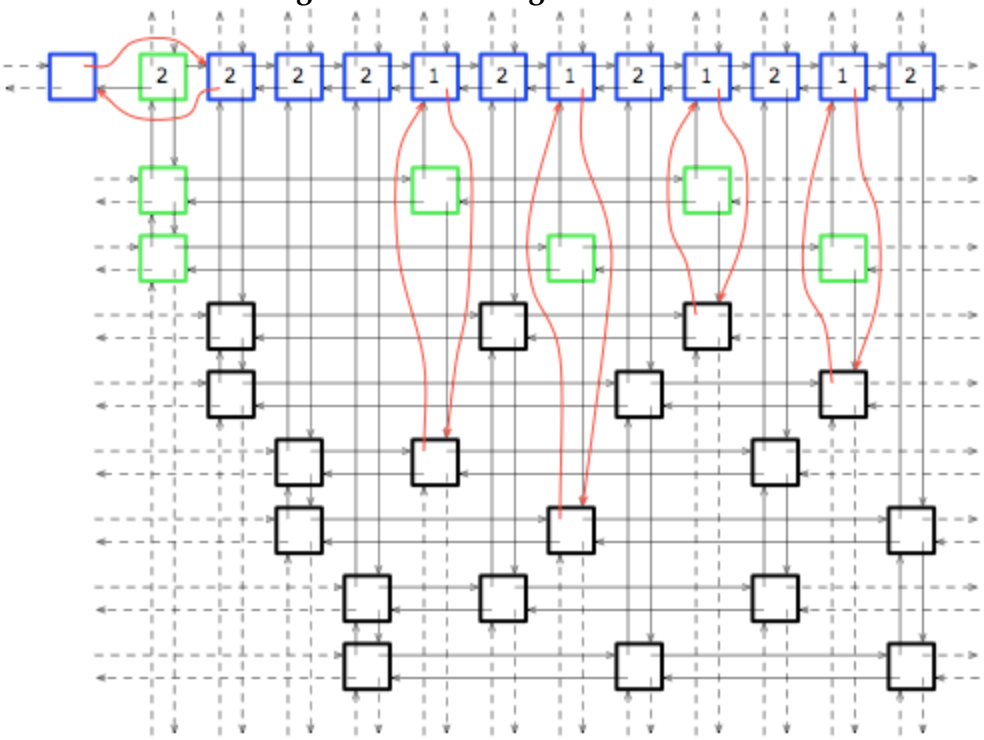
Figure 5 shows the initial setup of the matrix ready for searching. The blue cells are the column headers, each storing the number of cells in that column. The extra cell at the top left is the root of the whole data structure; it's there so that we can find the column headers of the constraints that we haven't satisfied yet. The dashed lines indicate links that wrap around the diagram.

Figure 5. Dancing Links matrix for the 2×2 Latin square.



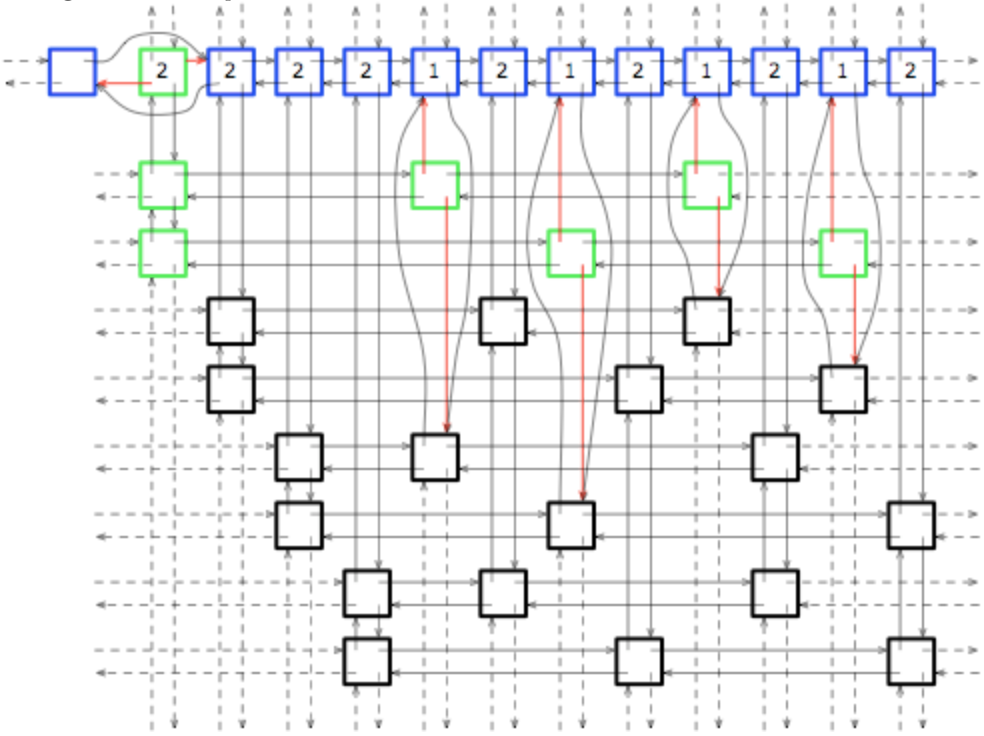
The basic operation is to remove a column from the matrix, together with all the rows that intersect that column. This operation can be used to implement step 1 and step 4. Knuth calls this "covering a column". **Figure 6** shows the matrix after covering the leftmost column. The green cells have been removed from the matrix; the red arrows show the links that have been altered.

Figure 6. "Covering" a column.



The key observation that makes Dancing Links efficient is that as long as you store a pointer to the column header, the "covering" operation is easy to reverse. That's because the cells that have been removed from matrix (coloured green in **figure 6**) still have pointers to their neighbours; they can be used to reverse the operation; Knuth calls this "uncovering a column". In **figure 7** these pointers are marked in red.

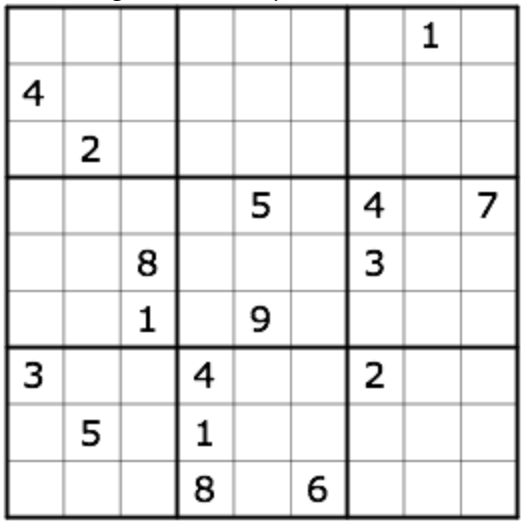
Figure 7. The pointers that can be used to "uncover" the column.



We need to take some care when backtracking to uncover the columns in precisely the reverse order to that in which we covered them. For the fiddly details, see Knuth's "**Dancing Links**" paper.

4.4 Application to sudoku

Figure 8. A tricky 9×9 sudoku.



The "Dancing Links" algorithm needs to run in two modes:

1. To generate a random solved grid. In this mode we stop the solver as soon as it reaches a solution. We want it to choose rows in a random order at step 2 of the algorithm, so that a good subset of the space of grids is reachable. We populate the top row of the grid with a random permutation of the nine symbols before running the solver. This gives a slight speedup—about 5%—over starting it on an empty grid.
2. To check a candidate puzzle to see if it has multiple solutions. In this mode we stop the solver when it finds a second solution (if any), or when it has examined the whole search tree and established that there is only a single solution. In this mode we don't care what order it chooses rows in at step 2, so we can save the effort of generating random numbers.

The "*S* heuristic" (choosing the column with the fewest rows) is very effective indeed. In a test of 100 randomly generated puzzles, the algorithm using the *S* heuristic used 345,810 search nodes to prove that each puzzle had a single solution. The same solver without the heuristic (always looking at the leftmost remaining column) needed 49,879,323 nodes to do the same.

This rough-and-ready test suggests that the *S* heuristic thus gives a speedup of about 144 times. But the distribution is quite wide. For easy puzzles, the speedup is often no more than 10 times. But some puzzles seem particularly tricky, for example when tackling the sudoku puzzle in **figure 8** (from Gordon Royle's **Minimum Sudoku** collection) the solver using the *S* heuristic uses only 64 nodes to prove that there is a single solution (no backtracking is necessary), while the solver without the heuristic needs 21,677,544 nodes, for a speedup of 338,711 times.

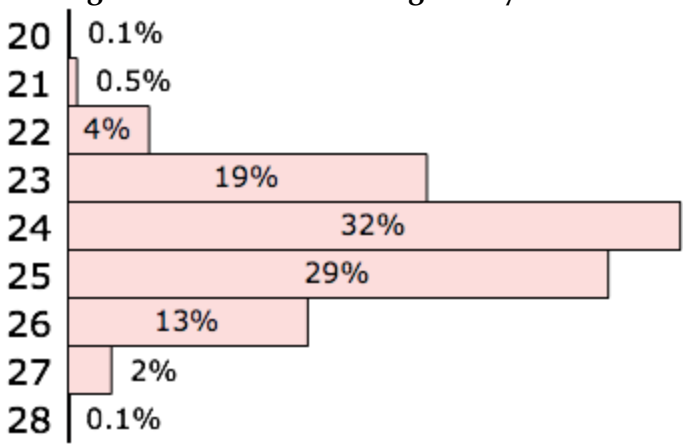
On the Nintendo DS, a rough rule of thumb was that it takes less than a frame ($^1/_{60}$ s) for the solver to prove that a candidate puzzle has a single solution.

4.5 Removing givens

The "Dancing Links" solver generates a solved sudoku grid. To make a puzzle, we remove symbols until removing any more would allow multiple solutions.

The table in **figure 9** shows the distribution of given symbols remaining after removing as many symbols as possible (based on 1,000 runs of the generator algorithm).

Figure 9. Distribution of given symbols.



5. Grader

Figure 10. A puzzle and its graded solution.

3	8 15	5 +7	6 22	4	2 21	9	1	7 23
4 +8	9	1 +10	8 ★40	5 47	7 39	6	2 3	3 48
2	6 16	7	3 56	1 12	9 57	8 +14	4 6	5 49
7 35	4	6	1	8 52	3 53	2 30	5 ★41	9 37
5 36	2 27	3 33	4	9 51	6 50	7 31	8 44	1
8	1 11	9 34	7 +38	2 32	5 42	4 5	3	6 43
1	7 25	2 26	9 55	3 54	4 4	5	6 45	8 46
9 29	3 28	4 9	5	6	8	1 2	7 +24	2
6 17	5 18	8	2 20	7	1 1	3 +13	9 +19	4

5.1 Overview

The design of the grader is quite straightforward. We have a repertoire of deduction tactics that we believe human solvers might use, ranked in order of their difficulty. At each stage the grader applies each of the tactics in turn, starting with the easiest, until one results in a symbol being written in. The difficulty of the hardest tactic used during the stage is recorded.

This process gives an order of solution of the puzzle, together with the difficulty of deducing each symbol. The difficulty of the whole puzzle is simply the difficulty of the hardest symbol. (We looked at more sophisticated measures of overall difficulty, such as the average of the difficulty of the deductions, but the evidence from playing the game is that no number of easy deductions can be the equivalent of a single hard deduction.)

Figure 10 shows a puzzle graded “hard”, with the given symbols in black and the symbols to be deduced in pink. The small numbers in the bottom right of each cell give the order in which the symbols were deduced by the grader; each “normal” move is annotated with a plus sign (+); each “hard” move is annotated with a star (★); and the remaining “easy” moves are plain.

5.2 Tactics

The tactics that the grader knows about are listed below. The names are by no means universally accepted, but are common on the web: see for example Simon Armstrong's collection of **solving techniques**.

1. Final symbol in set. A row, column, or block has eight symbols in it, which means that we can deduce the last one.
2. **Naked single**. A symbol which can only go into one of the cells of a row, column, or block.
3. **Hidden single**. A cell in which only one symbol can go.
4. **Naked pair**. Two symbols which can only go into two of the cells of a row, column, or block.
5. **Hidden pair**. Two cells into which only two symbols can go.
6. **Naked triple**. Three symbols which can only go into three of the cells of a row, column, or block.

A look through the literature on solving sudoku shows a number of other kinds of deductions with exotic names like "**x-wing**" and "**swordfish**". However, these are generally difficult deductions suited to pencil-and-paper puzzles. For the faster-paced pencil-less gameplay of *Zendoku*, it would not be fun if players were required to make these kinds of deductions. Indeed, even with this small set of deductions, at the "expert" level of difficulty it's probably **just as effective to guess the difficult moves** and accept the penalty for guessing wrong as it is to actually try to deduce the move.

If you were composing pencil-and-paper puzzles using this generate-and-test approach, you would want a much larger set of tactics.

5.3 What if the difficulty isn't right?

The grader gives us an estimate of how difficult the puzzle is. So what if the puzzle is the wrong difficulty, for example if we want an "easy" puzzle but the grader tells us we have a "normal" puzzle in hand? If the puzzle is too easy, we just throw it away and ask the generator for another. But if the puzzle is too hard, there's a neat way to make it easier. The grader gives us the difficulty of deducing each symbol, so we can simply fill in all the symbols that are too hard for the difficulty level we want. (This might overshoot and make the puzzle easier than we want, so except in the case where we want an easy puzzle, we still have to check the result. But that's cheaper than generating a new puzzle.)

Figure 11 shows the distribution of difficulty of puzzles as they emerge from the generator (based on 1,000 runs of the generator algorithm).

Figure 11. Distribution of grades.

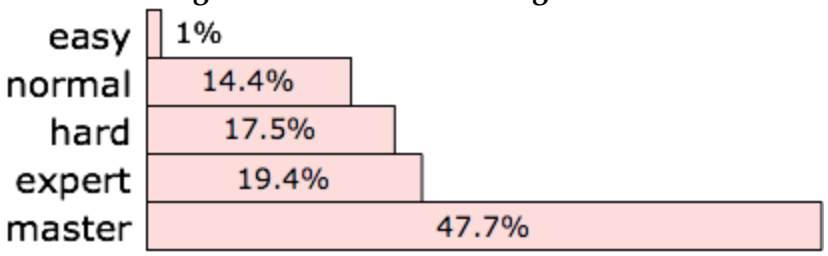
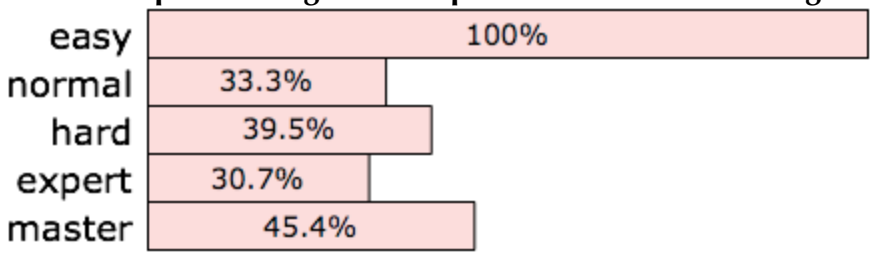


Figure 12 shows how successful the difficulty adjustment is, by giving the proportion of generated puzzles that can be converted to each dificulty level by filling in symbols that are too hard for that level (based on 1,000 runs of the generator algorithm).

Figure 12. Proportion of generated puzzles suitable for each grade.



There's a final wrinkle. It turns out that it's tricky to make fine enough distinctions at the easy end of the difficulty spectrum. When we playtested the game, it became apparent that puzzles graded "easy" spanned quite a wide range of difficulties, even though the deductions were much the same from one puzzle to another. (By that stage, we the developers had solved so many sudokus in the course of development that we no longer had any useful intuition about the difficulty of the puzzles!) It seems that the crucial factor is the "width" of the puzzle: how many deductions are available at a given time. When there are lots of available deductions, it's usually straightforward to find one of them quickly. When there are few, it takes longer to find them and the puzzle seems difficult.

We could have incorporated the number of available deductions into the grading calculations. But with little time remaining for testing, we found a quick but inelegant hack that seemed to do a good job of fixing the problem. Since more deductions become available as you enter more symbols, we specify for each difficulty level a minimum number of given symbols and simply enter symbols to the puzzle until the minimum number is reached. This results in puzzles that wouldn't pass muster in some newspapers—because some of the given symbols are deducible from some of the others—but do just fine in a video game.

6. Speed

Figure 12 shows that we may need to run the generation and grading algorithms several times before hitting on a puzzle that's of the right difficulty level. On the Nintendo DS this may take several seconds, so it looks as if we are in some danger of failing **requirement 4**.

To avoid the player having to wait, we arrange to keep a stock of puzzles at each difficulty in the saved data. When the player starts playing one of these puzzles, we start generating a new puzzle of that difficulty to fill the vacant slot. This generation can take place in a low-priority background thread while the game is being played.

7. Other uses for the grader

The grader takes a sudoku puzzle and returns a suggested move order together with the difficulty of each move. This algorithm comes in useful in two other places in the game.

In *Zendoku*'s solo modes, the grader is run in the background when the player makes a move. The first few moves in the suggested move order are displayed as hints. Note that we can't just look at the move order for the original puzzle, as we have to take into account the moves that the player has made so far. If the player has made a wrong move, we don't want to give this away by offering incorrect hints or failing to offer hints any more.

In the story mode, the grader is run in the background to generate the next move for the AI. The AI makes the first move suggested by the move order, taking time based on a function of the difficulty of the move and the "skill" of the AI. Again, we can't just let the AI use the original move order for the puzzle, because the player may have "fumbled" (made a mistake), resulting in a gift of symbols to the AI. The AI's play needs to take into account these extra symbols, as they will have made certain moves unnecessary, and other moves easier.

8. Competitors

Zendoku was not the first sudoku game for a handheld console to have puzzle generation capabilities; we were beaten to the punch by UFO Interactive's ***Sudoku Mania***, whose "auto-generating algorithm allows for an infinite number of puzzles".

Other competitors include:

- Nintendo's ***Brain Age***: "more than 100" puzzles.
- Hudson's ***Sudoku Gridmaster***: "more than 400" puzzles.
- D3's ***Essential Sudoku DS***: 1,000 puzzles.
- Ubisoft's ***Platinum Sudoku***: "20,000,000 different grids". It's hard to know what to make of this claim. These levels surely can't all be stored on the cartridge, it would be absurdly expensive. On the other hand if they were storing a smaller set of grids and permuting them, the number is absurdly small since the group of sudoku-preserving symmetries has size $2\times3!^8\times9!$ which is more than 3×10^8. Perhaps this has a generation algorithm too and the number is marketing-speak (like the "3 billion combinations" of the first Rubik's Cubes).

Of course, *Zendoku* has many unique features not found in other sudoku games…