Generating and Solving Sudoku Puzzles using DLX and SAT Solvers
CSC 428A - Spring 2014

Samuel Maskell
V00700592

April 23, 2014

Abstract

The constraint satisfaction nature of the Sudoku problem allows for the problem to be modelled as other constraint satisfaction problems such as exact cover and SAT. This paper discusses the use of the DLX algorithm for exact cover and Donald Knuth's SAT solvers for solving Sudoku puzzles. The work shows that DLX is ideal for generating Sudoku puzzles while the SAT approach is faster at solving well-formed puzzles.

1 The Game of Sudoku

The game of Sudoku involves a 9 by 9 grid made up of nine 3 by 3 subgrids, often referred to as blocks. Initially, certain cells contain a digit between 1 and 9 while others are left blank. The objective of the game is to fill in the remaining cells in such a way that each number between 1 and 9 appears exactly once in every row, every column, and every block[1]. This can be accomplished using a variety of techniques to infer the possible candidates for a given cell, removing possibilities until only one remains. However, in the case of a computer, it is generally simpler to apply a simple backtracking technique.

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

Unsolved Sudoku

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

Solved Sudoku

Figure 1: An Example Sudoku Puzzle

Given the nature of the problem, solving Sudoku puzzles can easily be modelled as many different constraint satisfaction problems, such as exact cover[2] or boolean satisfiability[1]. In doing so, standard techniques for solving these well known problems can be applied rather than creating a customized solution.

2 Sudoku as an Exact Cover Problem

Given a binary matrix, the exact cover problem is to find a set of rows such that there is a 1 in every column in exactly one of the rows in the set. Typically, the columns are used to represent constraints that must be satisfied. Each row in the matrix represents a partial solution, with conflicting partial solutions containing a 1 in the same column for the rule(s) on which they conflict[3].

Sudoku has four types of constraints

Cell Constraints Each cell must have exactly one number in it

Row Constraints Each row much contain each number exactly once

Column Constraints Each column much contain each number exactly once

Block Constraints Each block much contain each number exactly once

As such, the base set of constraints result in $4 * 9 * 9$ columns; there is a cell constraint for each of the 81 cells and there is a row, column and block constraint for each number in each row, column or block. Furthermore, there are $9 * 9 * 9$ rows, one for each number in each cell. The rows have 1's in the columns for the constraints that they satisfy[2].

Once the initial constraints have been set up, the columns corresponding to the constraints which are satisfied by any of the given cells in a puzzle can be removed, as well as all rows that could have satisfied any of those constraints. The remaining columns will be those which need to be satisfied in order to complete the puzzle and the rows will be the possible options.

3 Algorithm X and Dancing Links

In his paper, "Dancing Links", Donald Knuth describes an algorithm for solving the exact cover problem, which he calls Algorithm X, and a possible implementation technique, which he calls dancing links. This imple-

mentation is often referred to as DLX for short[3].

The dancing links approach is based on the observation that an element, x, in a doubly linked list can be removed in the following manner

$$x.left.right \leftarrow x.right, x.right.left \leftarrow x.left$$

and, more interestingly, x can be re-inserted into the list thusly

$$x.left.right \leftarrow x, x.right.left \leftarrow x$$

The matrix of the exact cover problem can be represented as a 2-dimensional, circular linked list with a node for every 1 in the matrix, with every node having up, down, left and right pointers. Knuth also uses a set of column header nodes that keep track of additional information about a given column as well as a special h node that simply points to the top of the first column. Furthermore, every node has a reference to its column header. This is data structure is shown in Figure 2. In the figure, the columns are labelled A through G and the numbers below the labels simply refer the number of rows in that column.

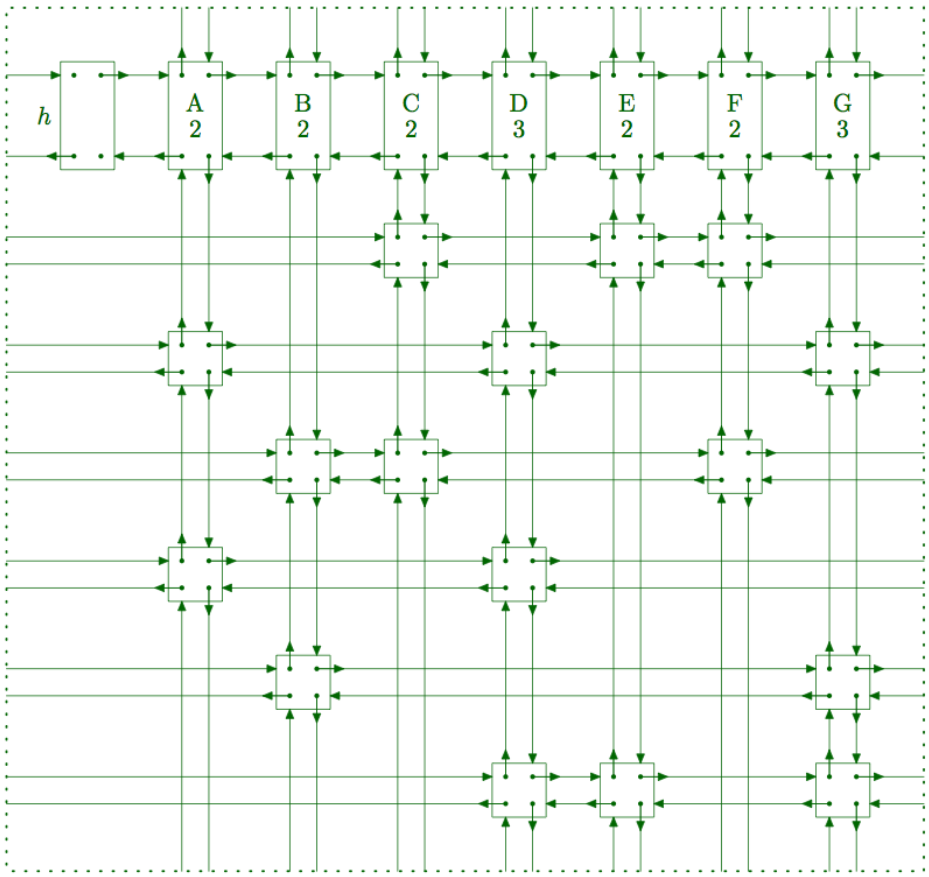

Figure 2: DLX data structure

Given this data structure, it is very easy to remove columns and rows, as well as to add them back in again, using the approach described above. The only constraint is that the columns/rows must be added back in in the opposite order from that in which they were removed. With this in mind, Algorithm X becomes quite simple.

Algorithm 1 DLX

```
procedure SEARCH(h: Node)
    if h.right = h then
        print solution
        return
    col ← CHOOSENEXTCOLUMN(h)
    COVER(col)
    for each row ← col.down, col.down.down, ..., while row ≠ col do
        add row to current solution
        for each j ← row.right, row.right.right, ..., while j ≠ row do
            COVER(j.colHeader)
        SEARCH(h)
        for each j ← row.left, row.left.left, ..., while j ≠ row do
            UNCOVER(j.colHeader)
        remove row from the current solution
    UNCOVER(col)
```

The idea is quite simple; we select a column that we want to satisfy and remove it and then, for each row that satisfies the chosen column, we try adding it to our current solution and removing all other columns that it satisfies and then we recurse. When the recursion completes, we must undo what we have done so far before we move on to the next candidate row.

The order in which the columns are chosen does not affect the final result. However, the amount of branching can be reduced if the column with the fewest number of rows is selected at each step.

The operation of "covering" is also fairly straight forward. We must simply remove the column header from list, so that it will not be chosen later, and then all the rows in the column from all other columns in which they appear. Uncovering, of course, is simply the exact opposite process, done in reverse order.

```
procedure COVER(col: Node)
    col.right.left ← col.left
    col.left.right ← col.right
    for each row ← col.down, col.down.down, ..., while row ≠ col do
        for each j ← row.right, row.right.right, ..., while j ≠ row do
            j.up.down ← j.down
            j.down.up ← j.up
            j.colHeader.size ← j.colHeader.size − 1
```

```
procedure UNCOVER(col: Node)
    for each row ← col.up, col.up.up, ..., while row ≠ col do
        for each j ← row.left, row.left.left, ..., while j ≠ row do
            j.up.down ← j
            j.down.up ← j
            j.colHeader.size ← j.colHeader.size + 1
    col.right.left ← col
    col.left.right ← col
```

4 Sudoku as a SAT Problem

The boolean satisfiability problem, SAT, involves finding a satisfying assignment for a boolean expression. These expressions are, typically, written in conjunctive normal form, CNF, i.e. a conjunction of a set of disjunctive clauses[1]. As this is a very well understood problem, there are a wide array of efficient SAT solvers that can be used to solve problems that can be represented as a SAT problem, such as Sudoku.

Similar to the exact cover problem, Sudoku lends itself well to being represented as a SAT problem. In order for a CNF expression to be satisfied, each clause must be satisfied, and in order of a clause to be satisfied, one of its literals must be true. As such, we can create a clause for each constraint, made up of literals that represent all of the possible ways to satisfy this constraint. However, we need to be careful to ensure that we do not end up with conflicting literals both being set to True in a satisfying assignment. This can be accomplished by carefully picking which constraints are

used or by simply including all possible constraints[1].

The variables of a Sudoku problem are simply the possible entries in each of the cells. As such, there must be a variable $s_{xyz} \forall x, y, z \in [1, 9]$ where x is a row, y is a column and z is the value in the cell which lies in both row x and column y. With this, we can build up clauses for each of the 4 necessary constraints.

Every cell has at least one number

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$$

Each number appears at most once in each row

$$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\overline{s_{xyz}} \vee \overline{s_{iyz}})$$

Each number appears at most once in each column

$$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (\overline{s_{xyz}} \vee \overline{s_{xiz}})$$

Each number appears at most once in each block

$$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{2} \bigwedge_{k=y+1}^{3} (\overline{s_{(3i+x)(3j+y)z}} \vee \overline{s_{(3i+x)(3j+k)z}})$$

$$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{2} \bigwedge_{y=1}^{3} \bigwedge_{k=x+1}^{3} \bigwedge_{l=1}^{3} (\overline{s_{(3i+x)(3j+y)z}} \vee \overline{s_{(3i+k)(3j+l)z}})$$

The above constraints are all that is necessary. However, each of these constraints has an inverse constraint that could also be added, which may or may not allow a SAT solver to eliminate clauses more quick and, thus, solve the puzzle more quickly.

Every cell has at most one number

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{8} \bigvee_{i=z+1}^{9} (\overline{s_{xyz}} \vee \overline{s_{xyi}})$$

Each number appears at least once in each row

$$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{x=1}^{9} s_{xyz}$$

Each number appears at least once in each column

$$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{y=1}^{9} s_{xyz}$$

Each number appears at least once in each block

$$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} s_{(3i+x)(3j+y)z}$$

The first 4 sets of constraints listed are usually known as the "minimal encoding" while the set of all 8 constraints is known as the "extended encoding". After all the constraint clauses are added, for either the minimal or extended encoding, we must then add clauses for each of the given elements in the initial grid. This clauses will only contain one literal, corresponding to the number and the row and column it appears in, forcing that variable to be true in any satisfying assignment.

5 Generating Sudoku Puzzles

In order to evaluate the different methods for solving Sudoku puzzles, a large set of puzzles is necessary for testing. Thankfully, the nature of the DLX algorithm makes it perfect for both solving and generating Sudoku puzzles. Given an empty starting grid, DLX will eventually generate all possible completed Sudoku puzzles. In order to randomize this process, so as to not generate a set of puzzles that are all closely related, we need only to randomize the order in which rows are selected at line 7 in Algorithm 1. The column selection need not be randomized as all columns

must be selected at some point before a valid solution is found. The order in which the rows are selected, however, define which way each constraint is satisfied[2].

In order to speed up the process slightly, a random permutation of 1-9 can be placed in the first row without needing to do any validation. The same idea can be applied to filling in the first column and the first block, by simply permuting the remaining available numbers. However, this process is not necessary for the algorithm to be able to generate a random puzzle.

After a completed grid is generated, we must remove a certain number of clues. A valid Sudoku puzzle must only have one solution[3] so a good strategy is to simply try removing each of the clues, i.e. currently filled-in cells, in a random order, testing to ensure that the puzzle still has only one solution at each step and replacing the clue if it has more than one. This can easily be accomplished using DLX as it is able to easily determine the number of possible solutions.

This approach typically results in very difficult puzzles being generated. In order to get around this, one could add clues back in until the desired difficulty is reached. However, this would require creating a system that can determine the difficulty of a Sudoku puzzle which would involve emulating the ways in which humans solve the puzzles. This was not necessary for the project, however, as difficult puzzles are a good test of the efficiency of solving algorithms.

6 Implementation

All source code is available on GitHub.

6.1 Sudoku Generation

In order to generate Sudoku puzzles, the Algorithm X was implemented in Python. This allowed for the algorithm to use of Python's generator construct. With this mechanic, the algorithm is very easily able to generate solutions to a given puzzle one at a time on demand, rather than generating all of them all at once, which is would be incredibly slow in the generation phase.

A noted by Ali Assaf of McGill University, Algorithm X can be very easily and succinctly implemented in python by using two dictionary objects, one which maps a column name to a set of rows and one which does the opposite[4]. Dictionary and set objects in Python have constant time insertion and deletion methods which means that Algorithm X can be efficiently implemented using a very similar technique to dancing links[5]. The dictionary and set data structure, however, allows for much simpler problem set-up. Furthermore, it is much simpler to permute a set in order to pick rows in a random order than it is to do so with a linked list. This makes the random generation process very straightforward.

6.2 Sudoku Solving

6.2.1 DLX

In order to truly test the efficiency of DLX on a Sudoku problem, Python is not a very suitable language as it requires much overhead. As such, DLX was implemented in C++, using structs to represent the nodes of the linked list. This implementation followed Knuth's description very closely. As the Sudoku generator was implemented using Algorithm X, it is capable of converting a Sudoku problem into an exact cover problem so it is used to do so. This representation of the problem is then written to standard out so that it can be read by the C++ program, which simply writes a list of row numbers of the solution to standard out. Unlike the Python implementation, the C++ version generates all possible solutions

to a given puzzle every time. However, since it is only used on valid Sudoku puzzles that only have one solution, this is not much of an issue.

As each row represents a certain digit being placed in a certain cell, it is very easy to map the result DLX, a list of rows, back to the Sudoku problem. One must simply fill in the cells that correspond to the rows in the solution to the exact cover problem in order to find the solution to the Sudoku puzzle.

6.2.2 SAT

As the Sudoku puzzles are generated in Python, another script for converting puzzles to SAT format was also written in Python. This conversion script is based heavily on the work done by Binnersley, Maskell and Simas[6]. This converted form is then written to standard out so that it can be read by a SAT solver. A collection of Knuth's SAT solvers, including SAT0W, SAT10 and SAT13, was used for solving the resulting SAT problem.

Each variable in the SAT problem correspond to a specific digit being placed in a certain cell. As such, one can find the solution to the Sudoku puzzle by filling in the cells corresponding to all of the unbarred literals in the satisfying assignment.

7 Results

Five hundred Sudoku puzzles were generated using the techniques described earlier. These were then converted to exact cover format and SAT, using both minimal and extended encodings. DLX and Knuth's SAT solver were then tested on the results. The average running times across the five hundred puzzles is listed in Table-1

Generator	DLX-All Sols	DLX-Single Sol	SAT13-Minimal	Sat13-Extended
0.4633	0.08678	0.085526	0.00628	0.007316

Table 1: Average running times in seconds for a single puzzle

Surprisingly, Knuth's SAT solver performed much better than DLX. However, it is clear than much time has been spent optimizing SAT13 so it is possible that better results could be achieved with DLX if it were to be optimized further. This is illustrated very clearly by the fact that Knuth's earlier SAT solvers, SAT10 and SAT0W, were also tested but were far too slow to complete the test set in any reasonable amount of time, taking many minutes to solve certain inputs.

Another surprising revelation is that DLX had almost identical performance whether it stopped after finding the first solution or not. This is most likely due to the fact that simply reading the input and building the data structure is the most expensive operation, and this is not affected by the change. If this process could be optimized, the algorithm would, most likely, run much faster.

SAT13 performed very well on both the minimal and extended encoding, though the minimal encoding gave better results. This is likely due to the fact that reading in the extended encoding and building the data structures takes much longer, as the input is roughly twice the size of the minimal encoding.

8 Conclusion

Given the nature of the Sudoku problem, it is very easy to map it to other constraint satisfaction problems. This allows for the use of well known algorithms such as DLX or certain SAT solving algorithms. Both of these approaches performed quite well, though Knuth's SAT13 yielded the best results. DLX, if slightly slower, however, is still ideal for Sudoku generation as it is able to easily generate all possible solutions to a Sudoku puzzle,

rather than simply stopping after finding the first one.

9 References

[1] I. Lynce and J. Ouaknine, "Sudoku as a SAT Problem," in ISAIM, 2006.

[2] G. Rees, "Zendoku puzzle generation," 10 June 2007. [Online]. Available: http://garethrees.org/2007/06/10/zendoku-generation. [Accessed 23 April 2014].

[3] D. Knuth, "Dancing Links," Millenial Perspectives in Computer Science, pp. 187-214, 2000.

[4] A. Assaf, "Algorithm X in 30 Lines," 2010. [Online]. Available: http://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html. [Accessed 23 April 2014].

[5] R. Pattis, "Complexity of Python Operations," 2014. [Online]. Available: https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt. [Accessed 23 April 2014].

[6] C. Binnersley, S. Maskell and B. Simas, "SAT-based Sudoku Solving," 2013.