TL; DR

Flip to the last page and there is a Sudoku solving example .

Documentation for DPLL SAT Solver and Sudoku

The DPLL procedure (with no pure literal elimination) is a direct translation of the pseudocode.

```haskell
import Data.List

type CNF = [[Int]]
type Valuation = [Int]

dpll :: CNF -> Valuation -> Valuation
dpll e v
    -- SAT
    | e == [] = v
    -- UNSAT
    | elem [] e = []
    -- Unit propagate if there are unit clauses
    | units /= [] = dpll (propagate unitLit e) (v ++ [unitLit])
    -- Propagate the first literal
    | otherwise = dpll (propagate lit e) (v ++ [lit]) >||< dpll (propagate (-lit) e) (v ++ [(-lit)])
        where
            -- Get the unit clauses
            units = filter (\x -> (length x) == 1) e
            -- Get the literal for unit propagation, only invoked if units is non-empty
            unitLit = head $ head units
            -- Get the first literal
            lit = head $ head e
            -- Propagation helper: first delete clauses then remove opposite polarity
            propagate n e = map (\\ [-n]) $ filter (notElem n) e
            -- Acts like ||
            (>||<) x y = if x /= [] then x else y
```

I am not going to explain too much about DPLL, INF1A course notes and Wikipedia have good information on it.

Unlike many DPLL implementation, my implementation always return a valuation instead of booleans (it returns an empty valuation if the CNF is unsatisfiable). This implementation is not safe nor is it good Haskell code. Though it is easy to read and understand.

One thing worth noting is that the first literal is always chosen, i.e. there isn't any heuristics, when there is no unit literal to propagate.

Input CNF Format

The CNF should first be converted to a simple data structure that closely resembles the standard DIMACS format.

Literals are represented with integers, the number is positive if the literal is positive and vice versa. So the following CNF:

$$\begin{gathered}(\neg A \vee B \vee C) \wedge \\ (A \vee C \vee \neg D) \wedge \\ (\neg B \vee \neg C \vee D)\end{gathered}$$

is converted to:

```
[[-1,  2,  3],
 [ 1,  3, -4],
 [-2, -3,  4]]
```

So the inner lists are considered to be connected conjunctively and the literals are connected disjunctively. Click here for more information on DIMACS format.

Example

A sample CNF is provided for you to test out the program:

```
e :: CNF
e = [[-1,  2,  3],
     [ 1,  3,  4],
     [ 1,  3, -4],
     [ 1, -3,  4],
     [ 1, -3, -4],
     [-2, -3,  4],
     [-1,  2, -3],
     [-1, -2,  3]]
```

Solve it in GHCi:

```
GHCi> :l DPLL
[1 of 1] Compiling DPLL             ( DPLL.hs, interpreted )
Ok, modules loaded: DPLL.
GHCi> dpll e []
[1,2,3,4]
```

I successfully made the computer solve Sudoku after reading these two resources:

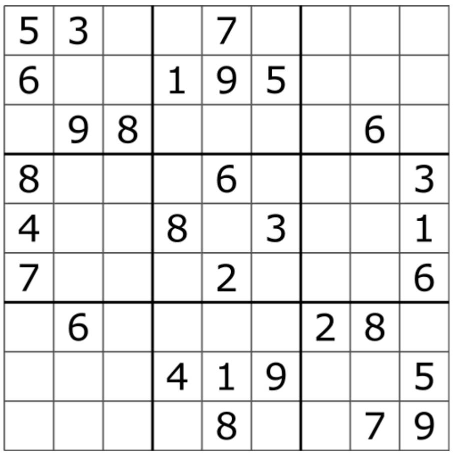
Sudoku SAT Encoding Description

Sudoku as a SAT Problem

Below is my best attempt at summarising those two amazing resources.

Sudoku Problem Description

Sudoku is a puzzle where you fill numbers in a 9x9 grid. Of course, you don't just fill in numbers randomly and there are some constraints. Below is a typical Sudoku puzzle:

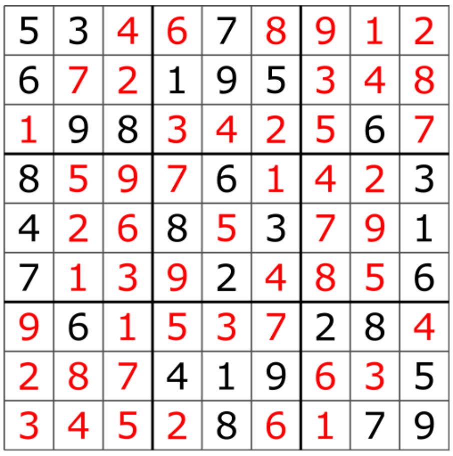


5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

You are given a partially filled grid and your task is to fill the whole grid abide by these constraints:

- Every cell must contain exactly one value from 1 to 9.
- Every row must contain all the numbers from 1 to 9.
- Every column must contain all the numbers from 1 to 9.
- Every block (those smaller 3x3 grids) must contain all the numbers from 1 to 9.

Below is the puzzle completed:



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

Encode Sudoku into CNF

We index the grid in the standard way where we use i to index the rows and j to index the columns, $i, j \in \{1,2,3,4,5,6,7,8,9\}$.

- ijn represents the proposition: the number n is filled in cell (i, j)
- $-ijn$ represents the proposition: the number n is **not** filled in cell (i, j)

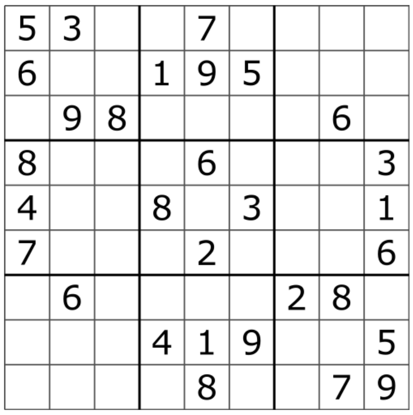
So 115 means the number 5 is filled in cell $(1, 1)$ and (-116) means the number 6 is not filled in the cell $(1, 1)$.

We can now start to encode all the constraints mentioned in the previous page into CNF:

- Every cell must contain exactly one value from 1 to 9. For cell $(1, 1)$, we need $(111 \vee 112 \vee 113 \vee 114 \vee 115 \vee 116 \vee 117 \vee 118 \vee 119)$ to assert that **at least** one number (from 1 to 9 obviously) is in cell $(1, 1)$. Also, we need to assert that it cannot be more than one number using $(\neg 111 \vee \neg 112) \wedge (\neg 111 \vee \neg 113) \wedge ... \wedge (\neg 118 \vee \neg 119)$. Do this for all 81 cells.
- Every row must contain all the numbers from 1 to 9. The encoding is extremely similar to the previous constraint. For row 1, we first assert that the number 1 must be in filled somewhere using $(111 \vee 121 \vee 131 \vee 141 \vee 151 \vee 161 \vee 171 \vee 181 \vee 191)$, and that if 1 is in one cell it cannot be in another using $(\neg 111 \vee \neg 121) \wedge (\neg 111 \vee \neg 131) \wedge ... \wedge (\neg 181 \vee \neg 191)$. Repeat this for all the numbers.
- Every column must contain all the numbers from 1 to 9.
- Every block (those smaller 3x3 grids) must contain all the numbers from 1 to 9.

The last two constraints are encoded in basically the same way, it is left to you as an exercise.

We just need to do one more thing before we feed the CNF into an DPLL solver. We need to encode the prefilled cells which is super simple. For the example Sudoku:



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

We just need $115 \wedge 123 \wedge 157 \wedge 216 \wedge ... \wedge 999$.

Observe that the 4 constraints (constrainsts excluding the pre-filled cells) is always the same for all the Sudoku puzzles and if you have encoded everything correctly there should be 11,988 clauses of which 324 are nine-ary and are 11,664 binary. There should also be 729 variables.

Below is the Haskell code:

```haskell
type Grid = [[Int]]
type CNF = [[Int]]

enc :: Grid -> CNF
enc g = prefilled g ++ encCll ++ encRow ++ encCol ++ encBlk

prefilled :: Grid -> CNF
prefilled g = [[(read (show i ++ show j ++ show n) :: Int)] |
               (i, row) <- zip [1..] g,
               (j, n) <- zip [1..] row,
               n /= 0]

encCll :: CNF
encCll = let
            a = [[read (show i ++ show j ++ show n) :: Int | n <- [1..9]]
                 | i <- [1..9], j <- [1..9]]
            b = [[-x, -y] | row <- a, (x:xs) <- tails row, y <- xs]
         in
            a ++ b

encRow :: CNF
encRow = let
            a = [[read (show i ++ show j ++ show n) :: Int | j <- [1..9]]
                 | i <- [1..9], n <- [1..9]]
            b = [[-x, -y] | row <- a, (x:xs) <- tails row, y <- xs]
         in
            a ++ b

encCol :: CNF
encCol = let
            a = [[read (show i ++ show j ++ show n) :: Int | i <- [1..9]]
                 | j <- [1..9], n <- [1..9]]
            b = [[-x, -y] | row <- a, (x:xs) <- tails row, y <- xs]
         in
            a ++ b

encBlk :: CNF
encBlk = let
            a = [[read (show i ++ show j ++ show n) :: Int | i <- [ib..ib + 2], j <- [jb..jb + 2]]
                 | ib <- [1,4,7], jb <- [1,4,7], n <- [1..9]]
            b = [[-x, -y] | row <- a, (x:xs) <- tails row, y <- xs]
         in
            a ++ b
```

`enc` connects all the sub-encodings together. It is better to put the constraints for those pre-filled cells in the beginning so that our naive DPLL algorithm runs faster.

Example

So given a Sudoku grid, we first encode it into CNF, then feed it into a DPLL solver.

Here is the example grid in Haskell:

```
exampleGrid :: Grid
exampleGrid = [[5,3,0, 0,7,0, 0,0,0],
               [6,0,0, 1,9,5, 0,0,0],
               [0,9,8, 0,0,0, 0,6,0],

               [8,0,0, 0,6,0, 0,0,3],
               [4,0,0, 8,0,3, 0,0,1],
               [7,0,0, 0,2,0, 0,0,6],

               [0,6,0, 0,0,0, 2,8,0],
               [0,0,0, 4,1,9, 0,0,5],
               [0,0,0, 0,8,0, 0,7,9]]
```

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

Encode it with `enc` (it is way too long to show the whole thing, only the length of the encoding is shown):

```
GHCi> length $ enc exampleGrid
12018
```

There are 12018 clauses.

Solve the CNF with DPLL:

```
GHCi> dpll (enc tGrid) []
[115,123,157,216,241,259,265,329,338,386,418,456,493,...
```

The result is too long to show on screen. DPLL only gives a satisfiable valuation but it doesn't construct the grid back. I wrote some small functions to do that and I encourage you to try it out yourselves.

The Finale

Download Sudoku.hs, Encode.hs, and DPLL.hs from the repository.
Open GHCi and type:

```
GHCi> :l Sudoku
[1 of 3] Compiling Encode           ( Encode.hs, interpreted )
[2 of 3] Compiling DPLL             ( DPLL.hs, interpreted )
[3 of 3] Compiling Main             ( Sudoku.hs, interpreted )
Ok, modules loaded: DPLL, Encode, Main.
GHCi>
```

We will solve the example Sudoku (You can also make your own using this format):

```
exampleGrid :: Grid
exampleGrid = [[5,3,0, 0,7,0, 0,0,0],
               [6,0,0, 1,9,5, 0,0,0],
               [0,9,8, 0,0,0, 0,6,0],

               [8,0,0, 0,6,0, 0,0,3],
               [4,0,0, 8,0,3, 0,0,1],
               [7,0,0, 0,2,0, 0,0,6],

               [0,6,0, 0,0,0, 2,8,0],
               [0,0,0, 4,1,9, 0,0,5],
               [0,0,0, 0,8,0, 0,7,9]]
```

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

```
GHCi> solve exampleGrid
[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]
(1.21 secs, 1,136,530,280 bytes)
```

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

Voilà!

Notice that it used 1,1 GigaBytes of ram to solve this so you need to ensure you have more than this amount. Alternatively you can head to the reple.it page and do:

```
> :l Sudoku
[1 of 3] Compiling DPLL             ( DPLL.hs, interpreted )
[2 of 3] Compiling Encode           ( Encode.hs, interpreted )
[3 of 3] Compiling Main             ( Sudoku.hs, interpreted )
Ok, three modules loaded.
(0.25 secs,)
```

```
=> (0.00 secs,)
> solve exampleGrid
[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]
(9.44 secs, 1,135,975,672 bytes)
=> (0.02 secs, 28,984 bytes)
```