# Sudoku Generation Using Human Logic Methods

February 17, 2008

**Abstract**

With the recent popularization of Sudoku, puzzle generation has become a matter of hobby to some, and obsession to others. Assuring valid puzzles with unique solutions while maintaining a consistent metric of difficulty is an arduous task. We present a set of metrics to assure a consistent difficulty level and methods that force unique solutions. Our model uses a search tree, however we outline several possibilities for heuristic approaches to minimize the time taken to produce a puzzle. We consider different branching techniques and different methods to seed an empty board with in an attempt to shortcut deep into the search tree.

# Contents

# 1 Introduction

Classic Sudoku is a logic puzzle consisting of a nine by nine grid in which each row, column, and three by three box consist of the digits one through nine. They are a special type of Latin Squares, which normally do not have the box requirement. For normal Latin Squares, determining whether or not a partially filled grid can be completed is known to be NP-complete[Garey and Johnson, 1979], or rather that polynomial time complexity algorithms are not known, and may not exist at all.

The popularity of Sudoku is relatively recent, sprouting up in 1984 in Japan by the Nikoli puzzle creating company and later in Europe in 2004 when The Times, a newspaper in London, decided to start publishing the puzzles. Following this, the puzzles have spread like wildfire across the globe.[Galanti, 2005]

Both the generation and rating of Sudoku puzzles have taken track in the fast lane after the recent popularization. People worldwide have quickly come to love the nine by nine constrained Latin Squares. Those who wish to produce puzzles have spent many hours devoted to the task, many to little avail. There is still much unknown about the seemingly simple logic puzzles, and only time will tell how far they will succumb to heuristics. Since Sudoku puzzles are a subset of Latin Squares, algorithms that apply to Latin Squares also apply to Sudoku. Since these are merely heuristics, it is possible that better algorithms exist, hidden somewhere in logic, diamonds in the rough.

The problem with determining difficulty is in how it is classified. Sudoku puzzles are generated by computers to be solved by people. Humans on the majority cannot keep track of the many branches of an exhaustive search or the overhead of Donald Knuth's Algorithm X that can be used to solve puzzles. Instead, most resort to logic and deductive reasoning. The complexity of this is what really determines the difficulty in solving a puzzle. There are many different techniques that separate the novice from the expert and by generating boards that require these advanced techniques, difficulty can be assured.

This is not only issue to resolve, however. Sudoku are generally considered invalid if they have multiple solutions, creating a complex intertwining of logical methods and uniqueness assurance. Mastering techniques that balance the two are the deciding factor in quality puzzle generation. Doing this in a timely matter is an even more perplexing problem.

## 1.1 Current Approaches

Puzzle makers have taken many different approaches to deciding what constitutes difficulty in a Sudoku board. Some do not try to manifest a puzzle of a certain difficulty, but rather generate puzzle after puzzle until one of the desired level is found. Some implementations do this completely randomly, while others follow perceived patterns. Some generate puzzle after puzzle and filter them through a sort of difficulty sieve, so that the designer has them at the ready when needed. This approach does not satisfy how we perceive the problem, in that we require creation of a board of exact difficulty on the spot.

## 1.2 Our Approach

We have developed a model to take several different approaches to the problem as we have interpreted it. To start, we obtained methods that would produce puzzles with difficulty similar to how we define our difficulty metric. Our model consisted of several logical techniques to apply to puzzles that a human would be capable of. By generating a database of puzzles using

this tool, we analyzed how far our methods progressed on them to check for consistency of difficulty. We performed analysis on this database to search for links between board difficulty and initial puzzle configuration, as can be seen in Appendix A.

Based on these findings and intuition, we set up our model to attempt to create puzzles. We employ a seed generation board through a series of different methods to try to identify patterns across difficulties. Following placement of the seed, the model uses several different logical approaches to solve puzzles in the same manner a human would and rates a given board based on the techniques it had to employ to solve it. This is coupled with a search tree which uses multiple methods of pruning. Leaf nodes of the tree are verified along the way to ensure uniqueness of a solution. It is possible for values inserted in the seed board to be removed and replaced in the search for a board the meets the required criteria. The complexity of this approach changes as the variables in our model change, though the dependancy on a search tree hinders the worst-case performance. This is further discussed in analysis of our results.

# 2 Sudoku Algorithms

In order to define levels of difficulty, we designed algorithms to implement several specific techniques used to solve sudoku puzzles. Six of these are based on methods of using human logic, and one is a machine-style implementation. Some of these algorithms do not assign values to cells on the board, instead eliminating possibilities and then allowing simpler methods to re-scan and actually assign values.

## 2.1 Human Methods

### 2.1.1 Naked Singles (ns)

Naked Singles iterates through the board, checking the number of possibilities in each cell. When it finds a cell with a single value as a possibility, it scans over the three regions containing that cell and removes that value from every list of possibilities that contains it.

### 2.1.2 Hidden Singles (hs)

Hidden Singles scans each region for the number of possibilities for each number ranging from one to nine. When it finds a cell containing a possibility which appears exactly once in a region, it assigns the cell the value of that possibility. It then removes that value from the list of possibilities for every region containing that cell.

### 2.1.3 Naked Pairs (np)

The first version of Naked Pairs searches each region for two values that occur only twice, and that share two cells, which may or may not contain other possibilities. Because they occur only in these two cells, one of them must go in one cell and the other in the second. Therefore, any other values in these two cells may be eliminated. The second version scans each region for two cells, each containing only the same two possibilities. Because one of these values must occur in each of the two cells, they cannot occur anywhere else in that region, and may be eliminated from the lists of possibilities for every other cell in the region.

### 2.1.4 Hidden Pairs (hp)

Hidden Pairs is not explicitly defined in our algorithm, but is instead implicitly implied by the combination of Naked Squares and Hidden Singles.

### 2.1.5 Intersection Removal (ir)

Intersection Removal implements two slightly different methods based on the same logic. First, it scans each region for all possibilities. If exactly two or three exist for a single number, several inferences may be made. If the possibilities occur on the same row in the same box, their values may be eliminated from the rest of the row. Similarly, if they are located in the same column and box, they may be deleted from the rest of the column. Second, if exactly two possibilities exist for a value in a row or column of a box, the value can be eliminated from the rest of the box.

### 2.1.6 X-Wing (xw)

X-wing searches for a single value that occurs exactly twice in two rows. If the cells containing the value line up in two columns to form a rectangle, all occurrences of the value may be removed from both columns. This also works the other way, starting with columns and eliminating from rows. Since this is one of the more advanced algorithms, we illustrate an example:

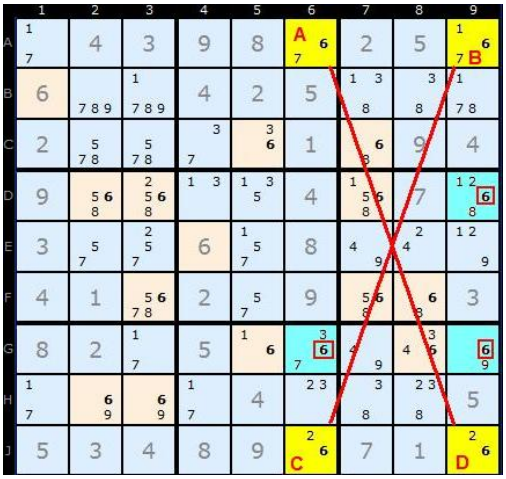


Figure 1: X-wing[Stuart, 2005]

Note that six is the only possibility in cells A6 and A9, as well as J6 and J9. A six will be forced into either opposite corner of the rectangle, assuring that a six is not possible anywhere else in the columns.

### 2.1.7 Y-Wing (yw)

Y-wings work by eliminating possibilities from intersections of influence. Each cell exerts a range of influence on all the others cells in the same row, column, and box. Y-wing is a complex, advanced technique, so we present an example:

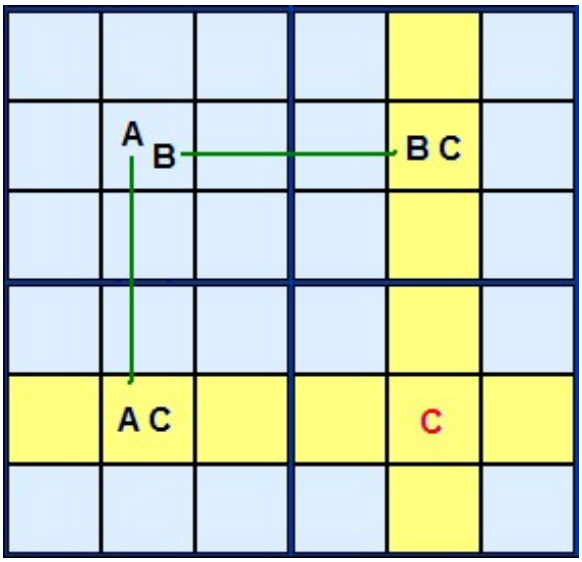


Figure 2: Y-wing[Stuart, 2005]

Suppose a cell has exactly two possibilities A and B. This cell AB is the pivot. Consider two cells, AC and BC, which are influenced by AB, but do not influence each other. If they each share exactly one possibility with AB and exactly one possibility with each other, then the possibility held in common between AC and BC, C, can be eliminated from every cell in the intersection of AC and BC's ranges of influence.

## 2.2 Machine Methods

### 2.2.1 Nishio

The method Nishio scans the whole board and locates the cell with the fewest remaining possibilities. It chooses one of these, assigns the value to the cell, and calls the human methods on the board. If this does not lead to a valid solution, it backs up and searches for the next cell with the fewest values.

## 2.3 Difficulty Metrics

Difficulty Strategies
Rank — Name — Required — Allowed

0 — Easy — ns — ns hs
1 — Medium — np — ns hs np
2 — Hard — ir — ns hs np ir
3 — Insane — xw xy — ns hs np ir xw xy

Difficulty Using these methods, we define five levels of difficulty: easy, medium, hard, insane, and computer:

- Easy: An easy puzzle is defined as one that can be solved using only Naked and Hidden singles, which keep track of only one cell or value at a time

- Medium: Medium puzzles require the use of Naked and Hidden Pairs, which search for patterns of two objects at the same time
- Hard: Hard puzzles require the use of at least one Intersection Removal, which tracks multiple objects and searches for patterns among them.
- Insane: Solving insane puzzles must require the use of X-wing or Y-wing, both of which track patterns of more than four objects.
- Computer: Computer level puzzles can only be solved by Nishio, which implements guessing and backtracking.

# 3 Exhaustive Search

Our model assures a unique solution because every time it branches, it assigns a value to the cell it is branching from. By saving the board state after the value is assigned, it is possible, when a solved state is encountered, to return to the saved state, ensuring that a single unique solution exists. Once a solved state is achieved, our model outputs a valid puzzle by backing up to the board state saved at the last branch. Because a valid solution was previously found beyond this state, it may be solved using only logical strategies.

It also assures a certain difficulty level is reached by calculating which logical game-playing strategies are used between the last branch and the solution. Certain strategies are required to attain a certain difficulty level. The rationale for grading a puzzle based on which methods are used is explained in the Difficulty Metrics section. If solving a puzzle fails to meet the criteria tabulated in the Difficulty Metrics section, the search backs up one level and explores other branches.

# 4 Conclusion

## 4.1 Time Complexity

Since our model employs variants of the exhaustive search method used for Exact Cover problems, the theoretical time complexity is expontential in the input size, in this case the size of the Sudoku board. At nine by nine this is kept relatively tame, but if we consider Sudoku variants produced in dimension equal to square numbers, the potential search tree explodes in size. The heuristical approach we have outlined does well in weeding out many of these branches by taking advantage of logical consequences, but this cannot overcome the exponential nature of the algorithm.

## 4.2 Initial Board State

Our model is flexible enough to consider several different initial board states. We took advantage of this, seeding the initial board with different amounts of initial hints. When done randomly, this was usually mostly worthless no matter what the depth. The search tree would have to backtrack, removing many of the initial hints given along with wasting time and computation. Following this, we would either start from an empty board, or seed a more thought out seed, which yielded better results.

### 4.3 Random Seed vs. Educated Seed

Through analysis of the puzzles generated by QQwing, as can be seen in Appendix A, we found that puzzles with unique solutions seemed to not have any variance across difficulties. What we have drawn from this is the fact that valid puzzles with a single solution seem to hover around starting values of three given in a row, col and bow, as well as of initial symbols given. By constructing a seed that had this type of configuration, the leaves of the search tree were more often found to be valid solutions, in which we could backtrack into unique solutions of the required difficulty. By centering around three for amounts given in rows, columns, boxes, and symbols, we believe this allows for many possible solutions, producing many viable options to be investigated instead of invalid leaves.

### 4.4 Solution Clusters

During execution of our model, we found that our solutions came quickly or not in a reasonable time frame. To investigate, instead of returning once a solution with the desired properties was found, our model instead traverses the tree further, recording each unique solution found along the way.

We found that solutions came in large clusters or not at all during the course of the experiment. The results of this experiment can be found in Appendix B.

## 5 Future Work

### 5.1 Symmetry

A great deal of symmetry is evident in a Sudoku grid. Consider the grid as a nine by nine matrix. Normally, we would be able to switch rows and columns while maintaining the nature of the original matrix. In a Sudoku puzzle, we have an additional constraint with boxes. This translate into three row and three column ”bands”, in which we can perform swaps as normal while still maintaining the integrity of the puzzle. In addition to this, bands can be swapped with other bands. The dependancies that govern the puzzle are maintained in each of these steps. Rotations and transpositions are also possible, though they overlap with each other (for instance, a sequence of transpositions results in a rotation.) In addition to these, the symbols used are not relevant to how the puzzle is solved, so we can swap any two numbers with each other. Logic is preserved throughout these operations, so if we consider six possible swaps for each row band and each column band, six permutations of each group of bands, and the number of ways we can order the digit one to nine, a single puzzle could produce

$$6^3 * 6^3 * 6^2 * 9!$$

possible equivalent puzzles. This does not take into consideration rotations and transpositions, but even still, a single puzzle potentially can be used to create over six hundred trillion equivalent puzzles. Boards that share this symmetry we consider to be in the same equivalence class of puzzles. While they require the same logic to solve, they are seemingly different to the naked eye. Using this and the fact that Sudoku puzzles of similar difficulty often require the same logical methods to solve, it is conceivable to collect a database of puzzles from different equivalence classes and permute a random puzzle for the user upon request. To further illustrate just how much a single puzzle can be modified while maintaining its integrity, we offer an example. Consider the following puzzles,

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A | | | 5 | | | | 3 | | |
| B | | 1 | | 5 | | | | 7 | |
| C | | | | | 3 | 1 | 6 | | 8 |
| D | 2 | | | | 8 | | 5 | | |
| E | 8 | | | | | | | | 9 |
| F | | | 7 | | 9 | | | | 2 |
| G | 9 | | 8 | 4 | 7 | | | | |
| H | | 4 | | | | 5 | | 6 | |
| I | | | 3 | | | | 4 | | |

(a) Original

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A | | | | | 5 | | | 2 | |
| B | | | 5 | | | 9 | 7 | | |
| C | 9 | 2 | | | | | | 4 | 8 |
| D | | 7 | 6 | 1 | 8 | | | | |
| E | 5 | | | | | 6 | 4 | | |
| F | | | | | 2 | | | 6 | |
| G | | 8 | | 3 | | | | 5 | |
| H | | | | 8 | | | | | 1 |
| I | | 1 | | | 7 | | | | 3 |

(b) Permuted

We obtain the permuted puzzle through the following operations applied to the original: Swap rows 1 and 3, 8 and 9, and 4 and 5. Swap columns 2 and 3, 4 and 6, and 7 and 8. Swap row bands 2 and 3, and column bands 1 and 2. Swap symbol 4 by 6, 1 by 9, 3 by 2.

The puzzles are seemingly quite different, but solutions require the exact same steps. A model that generates a puzzle would be hard pressed to ensure that upon repeated generations, the puzzle has not already been generated before. To effectively implement any sort of algorithm using these facts, a large database of puzzles from different equivalence classes would be required, and the generation of a single puzzle takes a lot of time and computational power to verify. There are, however, databases of such puzzles that have already been found.

### 5.1.1 Simulated Annealing

Since our model uses a search tree extensively, a Simulated Annealing approach may be possible. We considered this method, but failed to identify any patterns within different branches of the tree that produce favorable puzzles. Regions of influence are examined within select logical algorithms to identify how the removal of possibilties will affect the rest of the board, allowing the solver to draw conclusions. We believe there may be a pattern with how these regions of influence are affected by branches and that these patterns can be exploited. It may also be possible to do a breadth-first search instead of a depth-first search, looking one level down and seeing where our logical algorithms take us. This could be identified as a favorable state and traversed next, though as with depth-first methods, counterexamples that identify worst-case performance are likely.

### 5.1.2 Reverse Logic

It is conceivable that since our goal is to reach a finished puzzle from an initial state through logical conclusions, it will be possible to take the finished state of the board and work backwards in an attempt to artificially create the logic. We attempted this method, though the complexity after even trivial logical algorithms quickly exceeded what our model could handle. It does produce puzzles of the simplest complexity in little time, however. Through careful reversal of the algorithms, we believe it should be possible to produce more complex puzzles and is an area to be explored in the future.

# 6 References

# References

[Galanti, 2005] Galanti, G. (2005). The History of Sudoku.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

[Stuart, 2005] Stuart, A. (2005). Sudoku Strategies.

# 7 Appendecies

## 7.1 Appendix A

Distibution of values given in initial puzzles produced by QQwing

| $RowAmt$ | $Simple$ | $Easy$ | $Medium$ | $Hard$ |
|---|---|---|---|---|
| 0 | 882 | 671 | 759 | 644 |
| 1 | 3385 | 3449 | 3375 | 3182 |
| 2 | 6485 | 6933 | 7015 | 7022 |
| 3 | 7656 | 8180 | 8103 | 8332 |
| 4 | 5701 | 5630 | 5553 | 5635 |
| 5 | 2414 | 1847 | 1894 | 1917 |
| 6 | 461 | 281 | 294 | 255 |
| 7 | 16 | 7 | 7 | 13 |
| 8 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 |

| $ColumnAmt$ | $Simple$ | $Easy$ | $Medium$ | $Hard$ |
|---|---|---|---|---|
| 0 | 877 | 698 | 727 | 588 |
| 1 | 3382 | 3315 | 3426 | 3210 |
| 2 | 6498 | 6996 | 6929 | 7008 |
| 3 | 7605 | 8359 | 8253 | 8454 |
| 4 | 5773 | 5462 | 5464 | 5568 |
| 5 | 2424 | 1906 | 1910 | 1935 |
| 6 | 417 | 251 | 278 | 226 |
| 7 | 24 | 13 | 12 | 11 |
| 8 | 0 | 0 | 1 | 0 |
| 9 | 0 | 0 | 0 | 0 |

| $BoxAmt$ | $Simple$ | $Easy$ | $Medium$ | $Hard$ |
|---|---|---|---|---|
| 0 | 1554 | 980 | 961 | 817 |
| 1 | 3662 | 3584 | 3785 | 3545 |
| 2 | 5482 | 6595 | 6502 | 6676 |
| 3 | 6871 | 7698 | 7640 | 7780 |
| 4 | 5940 | 5617 | 5489 | 5606 |
| 5 | 2904 | 2149 | 2215 | 2165 |
| 6 | 553 | 361 | 387 | 395 |
| 7 | 34 | 16 | 21 | 16 |
| 8 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 |

| $SymbolAmt$ | $Simple$ | $Easy$ | $Medium$ | $Hard$ |
|---|---|---|---|---|
| 0 | 2986 | 2981 | 2974 | 2982 |
| 1 | 504 | 902 | 919 | 1043 |
| 2 | 4329 | 4808 | 4965 | 4761 |
| 3 | 10288 | 9923 | 9733 | 9248 |
| 4 | 7461 | 6938 | 6943 | 7247 |
| 5 | 1387 | 1368 | 1407 | 1643 |
| 6 | 45 | 80 | 59 | 76 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 |

## 7.2 Appendix B

| $Attempt$ | $NumofSols$ |
|---|---|
| 0 | 2519 |
| 1 | 2031 |
| 2 | 325 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 244 |
| 10 | 3501 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |
| 14 | 1630 |

| $Attempt$ | $NumNSUsed$ |
|---|---|
| 0 | 2519 |
| 1 | 2031 |
| 2 | 325 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 244 |
| 10 | 3501 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |
| 14 | 1630 |

| $Attempt$ | $NumHSUsed$ |
|---|---|
| 0 | 40 |
| 1 | 36 |
| 2 | 3 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 4 |
| 10 | 37 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |
| 14 | 16 |

| $Attempt$ | $NumNPUsed$ |
|---|---|
| 0 | 34 |
| 1 | 29 |
| 2 | 3 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 2 |
| 10 | 27 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |
| 14 | 12 |

| $Attempt$ | $NumIRUsed$ |
|---|---|
| 0 | 14 |
| 1 | 15 |
| 2 | 2 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 1 |
| 10 | 15 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |
| 14 | 6 |

| $Attempt$ | $NumXUsed$ |
|---|---|
| 0 | 12 |
| 1 | 18 |
| 2 | 1 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 2 |
| 10 | 5 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |
| 14 | 3 |

| $Attempt$ | $NumYUsed$ |
|---|---|
| 0 | 5 |
| 1 | 8 |
| 2 | 1 |
| 3 | 0 |
| 4 | 0 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 0 |
| 10 | 14 |
| 11 | 0 |
| 12 | 0 |
| 13 | 0 |
| 14 | 3 |

## 7.3 Appendix C: Code

file: SB.pm

```
package SB;
use warnings;
use strict;
use Storable qw/dclone/;
use Carp;
#use lib "$ENV{PREFIX}/lib/perl5";
#use Acme::Comment type => 'C++';
use List::MoreUtils qw/uniq/;
use List::Util qw/shuffle/;

sub new {
    my $class = shift;
    my %args = @_;
    my $self = bless {
        # The board references an array of arrayrefs that reference an array
        # storing the possibilities in a given cell.
        board => $args{board} || [
            map [ map [ 1 .. 9 ], 1 .. 9 ], 1 .. 9
        ],
        visited => [],
        choices => [],
        branch_save => [],
        totals => {},
    }, $class;
    return $self;
}

sub reset {
    my $self = shift;
    $self->{board} = [
        map [ map [ 1 .. 9 ], 1 .. 9 ], 1 .. 9
    ];
    $self->{visited} = [];
    $self->{choices} = [];
    totals => {},
}

sub revert_logic {
    my $self = shift;
    $self->{board} = pop @{ $self->{branch_save} };
}

# Read in a file
# The file may contain newlines or not
```

```
# Blanks are designated by characters that are not 1-9
sub read {
    my $self = shift;
    my $fname = shift;
    open my $file, "<", $fname or die "$fname: $!";
    $self->read_handle($file);
    close $file;
}

sub read_handle {
    my $self = shift;
    my $handle = shift;
    # Populate the board with the possibilities gleaned from reading a file
    $self->{board} = [
        map [
            map [
                # If the entry is a number from 1 to 9, store [ n ]
                # Otherwise if the entry is a "." or something, populate the
                # cell with the possibilities [ 1..9 ]
                /[1-9]/ ? $_ : 1 .. 9
            ], m/([^\r\n])/g # list of non-newline characters in the line
        ], <$handle>
    ];
    $self->is_valid or croak "Invalid board read in from the handle\n";
}

sub is_solved {
    my $self = shift;
    for my $y (0 .. 8) {
        for my $x (0 .. 8) {
            $self->get($x, $y) == 1 or return 0;
        }
    }
    return 1;
}

# Set a number for a given cell and recalculate the possibilities for the
sub set {
    my $self = shift;
    my ($x, $y, $n) = @_;
    ref $n eq "ARRAY" or croak "Not an array ref";
    $self->{board}[ $y ][ $x ] = $n;
}

# Eliminate the supplied list of possibilities from a given cell
sub eliminate {
    my $self = shift;
```

```
    my $x = shift;
    my $y = shift;

    my @eliminated;
    for my $n (@_) {
        # Don't eliminate the last value
        $self->get($x, $y) > 1 or return @eliminated;
        my @new = grep $_ != $n, @{ $self->{board}[ $y ][ $x ] };
        if (@{ $self->{board}[ $y ][ $x ] } != @new) {
            push @eliminated, $n;

            # DISPATCH CODE TO HIDDEN PAIRS, AND OTHERS? GOES HERE?
        }
        $self->{board}[ $y ][ $x ] = \@new; }
    return @eliminated;
}

# Whether a cell contains all of the supplied values as possibilities
sub contains {
    my $self = shift;
    my $x = shift;
    my $y = shift;
    for my $n (@_) {
        my $match = grep $_ == $n, @{ $self->{board}[ $y ][ $x ] };
        # Doesn't contain $n
        $match or return 0;
    }
    return 1;
}

# Get the possibilities for a given cell
sub get {
    my $self = shift;
    my ($x, $y) = @_;

#if( !$x and !$y ){print"invalid indices: x=$x, y=$y\n"; exit;}
    return @{ $self->{board}[ $y ][ $x ] };
}

# Print a board with the unknown values as "."s and the known values as
# themselves
sub print {
    my $self = shift;
    print join "\n", map {
        join "", map { @$_ == 1 ? @$_ : "." } @$_
    } @{ $self->{board} };
    print "\n";
```

```
}

sub print_saved {
    my $self = shift;
    print join "\n", map {
        join "", map { @$_ == 1 ? @$_ : "." } @$_
    } @{ $self->{branch_save}[-1] };
    print "\n";
}

# Return cells in a particular row
sub in_row {
    my $self = shift;
    my $n = shift;
    return @{ $self->{board}[ $n ] }[ 0 .. 8 ];
}

# Return known cells in a particular row
sub in_row_known {
    my $self = shift;
    my $n = shift;
    return map @$_, grep {
        ref eq "ARRAY" and @$_ == 1
    } @{ $self->{board}[ $n ] };
}

# Return unknown cells in a particular row
sub in_row_unknown {
    my $self = shift;
    my $n = shift;
    return grep {
        ref eq "ARRAY" and @$_ > 1
    } @{ $self->{board}[ $n ] };
}

# Return cells in a particular column
sub in_col {
    my $self = shift;
    my ($x, $y) = @_;
    my $n = shift;
    return map $_->[$n], grep ref eq "ARRAY", @{ $self->{board} };
}

# Return known cells in the same column as the given cell
sub in_col_known {
    my $self = shift;
    my $n = shift;
```

```
    return map @$_, grep {
        ref eq "ARRAY" and @$_ == 1
    } map $_->[$n], @{ $self->{board} };
}

# Return unknown cells in the same column as the given cell
sub in_col_unknown {
    my $self = shift;
    my ($x, $y) = @_;
    my $n = shift;
    return grep {
        ref eq "ARRAY" and @$_ > 1
    } map $_->[$n], @{ $self->{board} };
}

sub get_row_coords {
    my $self = shift;
    my $y = shift;
    return map [ $_, $y ], 0 .. 8;
}

sub get_col_coords {
    my $self = shift;
    my $x = shift;
    return map [ $x, $_ ], 0 .. 8;
}

# Return all the coordinates of the cells in the same box as the given cell
sub get_box_coords {
    my $self = shift;
    my ($x, $y) = @_;
    my $bx = 3 * int $x / 3;
    my $by = 3 * int $y / 3;
    my @coords;
    for my $cx ($bx .. $bx + 2) {
        for my $cy ($by .. $by + 2) {
            push @coords, [ $cx, $cy ];
        }
    }
    return @coords;
}

# Return all the cells in the same box as the given cell
sub in_box {
    my $self = shift;
    my ($x, $y) = @_;
    my $base_x = 3 * int $x / 3;
```

```
    my $base_y = 3 * int $y / 3;
    return map @{ $self->{board}[$base_y + $_] }[
        map $base_x + $_, 0 .. 2
    ], 0 .. 2;
}

# Return all the known cells in the same box as the given cell
sub in_box_known {
    my $self = shift;
    my ($x, $y) = @_;
    return map @$_, grep @$_ == 1, $self->in_box($x, $y);
}

sub in_box_unknown {
    my $self = shift;
    my ($x, $y) = @_;
    return grep @$_ > 1, $self->in_box($x, $y);
}

sub naked_singles {
  my $self = shift;
  my $changed = 0;
  for(my $x = 0;$x < 9;$x++) {
    for(my $y = 0;$y < 9;$y++) {
      my @possible = $self->get($x, $y);
      if(@possible == 1) {
        for(my $x2 = 0;$x2 < 9;$x2++) {
          if($x2 != $x) { if($self->eliminate($x2,$y,$possible[0])) { $changed
        }
        for(my $y2 = 0;$y2 < 9;$y2++) {
          if($y2 != $y) { if($self->eliminate($x,$y2,$possible[0])) { $changed
        }
        my @coords = $self->get_box_coords($x,$y);
        foreach my $c (@coords) {
          my $x2 = $c->[0];
          my $y2 = $c->[1];
          if($x != $x2 || $y != $y2) {
            if($self->eliminate($x2,$y2,$possible[0])) { $changed++; }
          }
        }
      }
    }
  }
  return $changed;
}

sub hidden_singles {
```

```
    my $self = shift;
    my $removed = 0; # number of possibilities removed

    for my $y (map 3 * $_, 0 .. 2) {
        for my $x (map 3 * $_, 0 .. 2) {
            my @box = $self->get_box_coords($x, $y);
            my %possible;
            for my $cell (@box) {
                $possible{"@$cell"} = join "", $self->get(@$cell);
            }
            for my $n (1 .. 9) {
                my @choices = grep $possible{$_} =~ m/$n/, keys %possible;
                if (@choices == 1 and length $possible{$choices[0]} > 1) {
                    # A number is only possible in one cell, set it
                    my ($cx, $cy) = split / /, $choices[0];
                    $self->set($cx, $cy, [ $n ]);
                    # Removed as many possibilities as the string'd get is lon
                    $removed += length $possible{ $choices[0] };
                }
            }
        }
    }
    return $removed;
}

sub naked_pairs {
    my $self = shift;
    my $elim_count = 0;

    my @boxes;
    for my $i (0, 3, 6) {
        push @boxes, map [ $self->get_box_coords($i, 3 * $_) ], 0 .. 2;
    }
    my @cols;
    for my $i (0 .. 8) {
        push @cols, [ map [ $_, $i ], 0 .. 8 ];
    }
    my @rows;
    for my $i (0 .. 8) {
        push @rows, [ map [ $i, $_ ], 0 .. 8 ];
    }
    my @regions = (@boxes, @cols, @rows);

    for my $region (@regions) {
        for my $n1 (1 .. 9){
            for my $n2 ($n1 + 1 .. 9) {
                my @matched;
```

```
                my $m1 = 0;
                my $m2 = 0;
                for my $cell (@$region) {
                    $m1 += $self->contains(@$cell, $n1);
                    $m2 += $self->contains(@$cell, $n2);
                    push @matched, $cell if $self->contains(@$cell, $n1, $n2);
                }
                if ($m1 == 2 and $m2 == 2 and @matched == 2) {
                    my @get_rid_of = grep { $_ != $n1 and $_ != $n2 } 1 .. 9;
                    $elim_count += $self->eliminate(
                        @{ $matched[0] }, @get_rid_of
                    );
                    $elim_count += $self->eliminate(
                        @{ $matched[1] }, @get_rid_of
                    );
                }

            }
        }

        my @pairs;
        for my $cell (@$region) {
            my $possible = join "", $self->get(@$cell);
            if (length $possible == 2) {
                push @pairs, $possible;
            }
        }
        for my $pair (@pairs) {
            if (2 == grep $pair eq $_, @pairs) {
                for my $c (grep join("", $self->get(@$_)) ne $pair, @$region)
                    $elim_count += $self->eliminate(@$c, split //, $pair);
                }
                last;
            }
        }
    }
    return $elim_count;
}


sub x_wing {
  my $self = shift;
  my $val = 0;
  for(my $x = 0;$x < 8;$x++) {
    for(my $y = 0; $y < 8;$y++) {
      my @a_pos = $self->get($x,$y);
      if(@a_pos > 1) {
```

```
foreach my $pos (@a_pos) {
  for(my $x2 = $x+2;$x2 < 9;$x2++) {
    #If we find a match in the row, let's look for a match in the col
    if($self->contains($x2, $y, $pos)) {
      for(my $y2 = $y+2;$y2 < 9;$y2++) {
        #Match here too?
        if($self->contains($x, $y2, $pos)) {
          #if the last one has it, we've found a box.
          if($self->contains($x2, $y2, $pos)) {
            my $count = 0;
            my $count2 = 0;
            #these need to be the *only* choices in two rows or two co
            for(my $i = 0;$i < 9;$i++) {
              $count += $self->contains($x,$i,$pos);
              $count2 += $self->contains($x2,$i,$pos);
            }
            #Are we the only possibilities in two cols? Then delete po
            if($count == 2 && $count2 == 2) {
              #print "Found A\n";
              for(my  $j = 0;$j < 9;$j++) {
                if($j != $x && $j != $x2) {
                  $val += $self->eliminate($j,$y,$pos);
                  $val += $self->eliminate($j,$y2,$pos);
                }
              }
            }
            else {
              $count = 0;
              $count2 = 0;
              for(my $i = 0;$i < 9;$i++) {
                $count += $self->contains($i,$y,$pos);
                $count2 += $self->contains($i,$y2,$pos);
              }
              if($count == 2 && $count2 == 2) {
                #print "Found B\n";
                for(my $j = 0;$j < 9;$j++) {
                  if($j != $y && $j != $y2) {
                    $val += $self->eliminate($x,$j,$pos);
                    $val += $self->eliminate($x2,$j,$pos);
                  }
                }
              }
            }
          }
        }
      }
    }
```

```
            }
          }
        }
      }
    }
    return $val;
  }

  sub int_removal {
    my $self = shift;
    my $val = 0;
    #my @coords = (1,5,9);
    my @coords = (0, 3, 6);
    #/*
    #$self->naked_singles();
    foreach my $x (@coords) {
      foreach my $y (@coords) {
        my $s = join "", map @$_, $self->in_box_unknown($x,$y);
        #print "$s\n";
        my @nums = (1 .. 9);
        foreach my $num (@nums) {
          my $n = length join "", $s =~ m/($num)/g;
          if($n == 2 || $n == 3) {
            #print "s: $s n: $n num: $num\n";
            my $total = 0;
            for(my $i = 0;$i < 2;$i++) {
              $total += $self->contains($x,$y+$i,$num);
            }
            #$n in the same col, delete everywhere else in col
            if($total == $n) {
              for(my $i = 0;$i < 9;$i++) {
                my $v = 0;
                if($i != $y && $i != $y+1 && $i != $y+2) {
                  if($self->eliminate($x,$i,$num)) { $v++; }
                }
                if($v) { return $val + $v; }
              }
            }
            $total = 0;
            for(my $i = 0;$i < 2;$i++) {
              $total += $self->contains($x+$i,$y,$num);
            }
            #$n in the same row, delete everywhere else in row
            if($total == $n) {
              for(my $i = 0;$i < 9;$i++) {
                my $v = 0;
                if($i != $x && $i != $x+1 && $i != $x+2) {
```

```
              if($self->eliminate($i,$y,$num)) { $v++; }
            }
            if($v) { return $val + $v; }
          }
        }
      }
    }
  }
  for(my $i = 0;$i < 9; $i++) {
    my $s = join "", map @$_, $self->in_row_unknown($i);
    #print "row $i: $s\n";
    foreach my $num (1..9) {
      my $n = length join "", $s =~ m/($num)/g;
      if($n == 2 || $n == 3) {
        my $total;
        for(my $j = 0;$j < 3; $j++) {
          $total = 0;
          for(my $k = 0;$k < 3; $k++) {
            $total += $self->contains(($j*3)+$k,$i,$num);
          }
          if($total == $n) {
            #print "total of $num (rows) is $total and n was $n\n";
            my @box = $self->get_box_coords($j*3,$i);
            my $v = 0;
            foreach my $c (@box) {
              #print "x: $c->[0] y: $c->[1]\n";
              if($c->[1] != $i) {
                if($self->eliminate($c->[0],$c->[1],$num)) { $v++; }
              }
            }
            if($v) { return $val + $v; }
          }
        }
      }
    }
  }
  for(my $i = 0;$i < 9; $i++) {
    my $s = join "", map @$_, $self->in_col_unknown($i,0);
    foreach my $num (1..9) {
      my $n = length join "", $s =~ m/($num)/g;
      if($n == 2 || $n == 3) {
        my $total;
        for(my $j = 0;$j < 3; $j++) {
          $total = 0;
          for(my $k = 0;$k < 3; $k++) {
            $total += $self->contains($i,($j*3)+$k,$num);
```

```
            }

            if($total == $n) {
              #print "total of $num (cols) is $total and n was $n\n";
              my @box = $self->get_box_coords($i,$j*3);
              my $v = 0;
              foreach my $c (@box) {
                if($c->[0] != $i) {
                  if($self->eliminate($c->[0],$c->[1],$num)) { $v++; }
                }
              }
              if($v) { return $val + $v; }
            }
          }
        }
      }

      return $val;
    }

    sub deadly_rects {
      my $self = shift;
      my @coords = (0,3);
      my @box = (0..2);
      foreach my $x (@coords) {
        foreach my $y (@coords) {
          for(my $i = 0;$i < 3;$i++) {
            for(my $j = 0;$j < 3;$j++) {
            }
          }
        }
      }
    }
    sub nishio {
      my $self = shift;

      my $min_poss = 10;
      my $x_coord = -1;
      my $y_coord = -1;
      my @poss;

      my @foo;
      for(my $y = 0; $y < 9; $y++){
        for(my $x = 0; $x < 9; $x++){
          @poss = $self->get($x,$y);
```

```
      #if(scalar(@poss) < $min_poss && scalar(@poss) > 1){
      #  $min_poss = scalar(@poss);
      if (@poss > 1) {
        push @foo, [ $x, $y ];
      }
    }
  }
#  for my $y (shuffle 0 .. 8) {
#    for my $x (shuffle 0 .. 8) {
#      my @possible = $self->get($x, $y);
#      next if @possible == 1;
#      $x_coord = $x;
#      $y_coord = $y;
#    }
#  }

  if(@foo == 0){
    if (!$self->is_solved and !$self->is_valid) {
       $self->back;
       return;
    }
  }
  else{
    ($x_coord, $y_coord) = @{ $foo[rand @foo] };
    my @new_poss = $self->get($x_coord,$y_coord);

    my $branch_value = $new_poss[int(rand(scalar(@new_poss)))];

    $self->branch($x_coord, $y_coord, $branch_value);

# print "depth = ", $self->get_depth, "\n";

  }
}

sub seed {
    my $self = shift;
    my $target = shift;
    # Fill in completely empty cells
    my $sanity = 0;
    my $found = 0;
    while ($found < $target) {
        my $x = int rand 9;
        my $y = int rand 9;
        $self->get($x,$y) == 9 or next;

        my $value = 1 + int rand 9;
```

```
            if ( $self->is_valid($x, $y, $value) ){
                $self->branch($x, $y, $value);
                $found ++;
            }
        }
    }

    sub is_valid {
        my $self = shift;
        if (@_ == 0) {
            for my $cy (0 .. 8) {
                for my $cx (0 .. 8) {
                    my @p = $self->get($cx, $cy);
                    @p == 1 or next;
    #print "is_valid($cx, $cy, $p[0]) = ", $self->is_valid($cx, $cy, $p[0]);
                    return 0 if $self->is_valid($cx, $cy, $p[0]) == 0;
                }
            }
            return 1;
        }

        my ($x, $y, $n) = @_;

        my @row_values = map $self->get(@$_), grep {
            # Known element and not at the same spot as is being checked
            ($x != $_->[0] or $y != $_->[1]) and $self->get(@$_) == 1
        } $self->get_row_coords($y);

        foreach my $rv (@row_values){
            return 0 if $n == $rv;
        }

        my @col_values = map $self->get(@$_), grep {
            # Known element and not at the same spot as is being checked
            ($x != $_->[0] or $y != $_->[1]) and $self->get(@$_) == 1
        } $self->get_col_coords($x);

        foreach my $cv (@col_values){
            return 0 if $n == $cv;
        }

        my @box_values = map $self->get(@$_), grep{
            # Known element and not at the same spot as is being checked
            ($x != $_->[0] or $y != $_->[1]) and $self->get(@$_) == 1
        } $self->get_box_coords($x, $y);

        foreach my $bv (@box_values){
```

```
            return 0 if $n == $bv;
        }

        # Didn't find anything wrong, must be ok, at least for the cell at the
        # provided (x, y)'s point of view
        return 1;
    }

    sub get_depth {
        my $self = shift;
        return scalar @{ $self->{visited} };
    }

    sub branch {
        my $self = shift;
        my ($x, $y, $value) = @_;
        push @{ $self->{choices} }, {
            x => $x,
            y => $y,
            value => $value,
        };
        $self->eliminate($x, $y, $value);

        # push copy, rest of possibilities
        my $copy = dclone($self->{board});
        push @{ $self->{visited} }, $copy;

        # reset totals
        $self->{totals} = {};

        $self->set($x, $y, [ $value ]);

        push @{ $self->{branch_save} }, dclone($self->{board});
    }

    # Back track and return the choice that was made at that level
    sub back {
        my $self = shift;
        $self->{board} = pop @{ $self->{visited} };

        # reset totals
        $self->{totals} = {};

        pop @{ $self->{branch_save} };
        return pop @{ $self->{choices} };
    }
```

```
sub y_wing {
    my $self = shift;
    my $elim = 0;
    for my $y (0 .. 8) {
        for my $x (0 .. 8) {
            my @p = $self->get($x, $y);
            @p == 2 or next;
            my @influence = $self->influence($x, $y);
            my @matches;
            for my $coord (@influence) {
                for my $n (@p) {
                    # Has two possibilities
                    $self->get(@$coord) == 2 or next;

                    # Contains one of the possibilities
                    $self->contains(@$coord, $n) or next;
                    # but not both
                    $self->contains(@$coord, @p) and next;

                    push @matches, $coord;
                }
            }
            @matches or next;
            for my $n (1 .. 9) {
                next if $self->contains($x, $y, $n);
                my @pincers = grep $self->contains(@$_, $n), @matches;
                @pincers or next;

                if (@pincers == 2) {
                    my $sg1 = join "", $self->get(@{$pincers[0]});
                    my $sg2 = join "", $self->get(@{$pincers[1]});
                    next if $sg1 eq $sg2;

                    # And not in the pincers box
                    my $bx = int $pincers[0][0] / 3 == int $pincers[1][0] / 3;
                    my $by = int $pincers[0][1] / 3 == int $pincers[1][1] / 3;
                    next if $bx and $by;

                    # And not in the pincers row or column
                    $pincers[0][0] != $pincers[1][0] or next;
                    $pincers[0][1] != $pincers[1][1] or next;
                    my @ii = $self->influence_intersect(map @$_, @pincers);
#print "($x, $y):", join("",$self->get($x, $y)),
#    " => ",
#    "(@{$pincers[0]}):", join("", $self->get(@{$pincers[0]})),
#    ", (@{$pincers[1]}):",join("", $self->get(@{$pincers[1]})),"\n";
#print "    ii=", join " ", map "[@$_]", @ii;
```

```
#print "\n";

                    # Don't delete the master cell
                    @ii = grep { $x != $_->[0] or $y != $_->[1] } @ii;
#print "    ii=", join " ", map "[@$_]", @ii;
#print "\n";

                    # Eliminate all cells in the ii that contain $n
                    # except for the single values
                    @ii = grep $self->get(@$_) > 1, @ii;
#print "    ii=", join " ", map "[@$_]", @ii;
#print "\n";
#print "    eliminate(@$_, $n):", $self->get(@$_), "\n" for @ii;
                    $elim += $self->eliminate(@$_, $n) for @ii;
                }
            }
        }
    }
    return $elim;
}

# Return all the coords that the given cell has an influence on
sub influence {
    my $self = shift;
    my ($x, $y) = @_;
    my @coords = (
        $self->get_box_coords($x, $y),
        $self->get_col_coords($x),
        $self->get_row_coords($y)
    );
    # Don't return the coordinate itself
    @coords = grep { ($_->[0] == $x and $_->[1] == $y) ? 0 : 1 } @coords;
    return map [ split / / ], uniq map join(" ", @$_), @coords;
}

# Return the intersection of two influences as an array of coords
sub influence_intersect {
    my $self = shift;
    my ($x1, $y1, $x2, $y2) = @_;
    my @inf1 = $self->influence($x1, $y1);
    my @inf2 = $self->influence($x2, $y2);

    # Intersection of the influences with hashes
    my %int;
    ($int{join " ", @$_} ||= 0) ++ for @inf1, @inf2;
    return map [ split / / ], grep $int{$_} == 2, keys %int;
}
```

```
# Progress through the puzzle by logic
# Returns whether anything happened
sub solve {
    my $self = shift;
    my @algos = @_; # which algorithms to use

    # push saved last branch thing
    #push @{ $self->{branch_save} }, dclone($self->{board});
    my %t;
    $t{$_} = 0 for @algos;

    my $anything = 0;
    my $working = 1; # whatever you're doing, it's working!
    while ($working) {
        $working = 0;
        for my $algo (@algos) {
            my $elim = 0; # how many were eliminated with this algorithm
            if ($algo eq "ns") {
                $elim = $self->naked_singles;
            }
            elsif ($algo eq "hs") {
                $elim = $self->hidden_singles;
            }
            elsif ($algo eq "np") {
                $elim = $self->naked_pairs;
            }
            elsif ($algo eq "ir") {
                $elim = $self->int_removal;
            }
            elsif ($algo eq "xw") {
                $elim = $self->x_wing;
            }
            elsif ($algo eq "yw") {
                $elim = $self->y_wing;
            }

            $self->{totals}{$algo} ||= 0;
            $self->{totals}{$algo} += $elim;
            $working = 1 if $elim > 0;
            redo if $elim;
        }
        $anything += $working;
    }
    #my $totals = $self->{totals}[-1];
    #print "totals ", join(", ", map "$_:$totals->{$_}", keys %$totals), "\n";
    #$self->print;
```

```
    #print "\n";
    return $anything;
}

1;

file: make_board.pl
#!/usr/bin/env perl
use warnings;
use strict;
$|++;

use SB;
my $sb = SB->new;

my $filename = shift;
open my $fh, ">>", $filename;
END {
    close $fh;
}

#my $seed = shift @ARGV || 0;
#if (-e $seed and not -d $seed) {
#    $sb->read($seed);
#    $sb->print;
#}
#else {
#    $sb->seed($seed);
#    $sb->is_valid or die "NOT VALID\n";
#}

my %levels = (
    sour => { # Gumbo chicken
        can_have => [ qw/ns hs/ ],
        must_have => [ qw/ns/ ],
    },
    mild => {
        can_have => [ qw/ns hs np/ ],
        must_have => [ qw/np/ ],
    },
    spicy => { # Spicy chicken
        can_have => [ qw/ns hs np ir/ ],
        must_have => [ qw/ir/ ],
    },
    cajun => { # Chicken chicken: chicken chicken chicken
        can_have => [ qw/ns hs np ir xw xy/ ],
        must_have => [ qw/xw xy/ ], # one or more of these
```

```
    },
#    insanity => qw//,
);

my $level = shift @ARGV || "mild";

searching: while (1) {
    $sb->solve(qw/ns hs np hs ir xw yw/);
#    $sb->solve(qw/ns hs np hs xw yw/);
    if (not $sb->is_valid) {
        # The board isn't valid, back track
        $sb->back;
print "Invalid, back tracking\n";
    }
    elsif ($sb->is_solved) {
        # Hit a solved state (leaf node)
        # See if the board conforms to the constraints

        print $fh "TOTALS ",
            join ", ", map "$_:$sb->{totals}{$_}", keys %{ $sb->{totals} };
        print $fh "\n";

        # Methods used to solve the last branch
        my @used = grep $sb->{totals}{$_}, keys %{ $sb->{totals} };
        print $fh "USED @used\n";
        print "SOLUTION (@used)\n";
#        my $must = 0;

#        for my $method (@used) {
#            if (grep $method eq $_, @{ $levels{$level}{must_have} }) {
#                $must = 1;
#            }
#            if (not grep $method eq $_, @{ $levels{$level}{can_have} }) {
#                # Method not allowed in the target difficulty level
#                $sb->back;
#                next searching;
#            }
#        }
#        if ($must == 0) {
#            $sb->back;
#            next searching;
#        }
        #last searching;
        $sb->back;
        next searching;
    }
    else {
```

```
        # Run nishio to generate a new branch
print "New branch (", $sb->get_depth, ")\n";
        $sb->nishio;
$sb->solve(qw/ns hs np hs ir xw yw/);
    }
}

#$sb->print;
#print "** PUZZLE **\n";
$sb->print_saved;

file: solve.pl
#!/usr/bin/env perl
use warnings;
use strict;
$|++;

use SB;
my $sb = SB->new;

my $puzzle = shift @ARGV || "4/";

my ($path, $file_num) = split m{:}, $puzzle;
#my @files = glob "../boards/qqwing/${level}_files/*";

$file_num ||= 0;
if ($file_num < 0) {
    $file_num = int rand grep /^\./, glob $path;
}

if (-d $path) {
    opendir my($dh), $path;
    for (my $i = 0; $i <= $file_num; $i++) {
        my $f = scalar(readdir $dh);
        redo if $f =~ m/^\./;
    }
    $path = "$path/" .readdir $dh;
    closedir $dh;
}

print "$path : $file_num\n";

my @algos = qw{ ns hs np ir xw yw };
@algos = @ARGV if @ARGV;
print "Using algorithms: @algos\n";

if ($path eq "-") {
```

```
    $sb->read_handle(*STDIN);
}
else {
    $sb->read($path);
}

$sb->is_valid or print "Not valid\n";

print "\n** INITIAL **\n";
$sb->print;
print "\n";

$sb->solve(@algos);

print "\n** FINAL **\n";
$sb->print;
print "\n";
$sb->is_valid or print "Not valid\n";
$sb->is_solved or print "Not solved\n";

print "Totals: ", join ", ",
    map "$_=$sb->{totals}{$_}", keys %{ $sb->{totals} };
print "\n";
```