Yet another sudoku in clojure

1 Introduction

I am quite new in functional programming and multiprocessing. I encountered clojure and reinstated a small project I used to get familiar with MIDP. This project provided sudoku solving and generation on small mobile phones.

This topic for me seemed to be quite fit to evaluate how the language and I might fit together. Therefore a port of said project was the goal.

During trying to understand clojure, the old code and seeing that the idea to port object-oriented to functional might not be the best of my ideas, I decided to let the old project only provide the internal representation of the sudoku-board and some rough ideas of solving and generation.

This document tries to give an overview over the supported usecases and the concepts behind.

sudoku in clojure, an ugly user-interface but capable to create really fiendish sudokus

2 A Small Sudoku-Kit

What has been created is a small and ugly program usable to:

- Let the computer solve sudokus
- Check a kind of calculated complexity of sudokus
- Create new sudokus as complex as possible in a given time using a given number of threads (aka processors)

2.1 Outline

If your are able to start the program, how this might become possible will be described later, you might see something like the following picture. As anybody might notice quite fast, the Userinterface has not been designed very thoroughly but developed with upcoming needs to implement a few usecases.

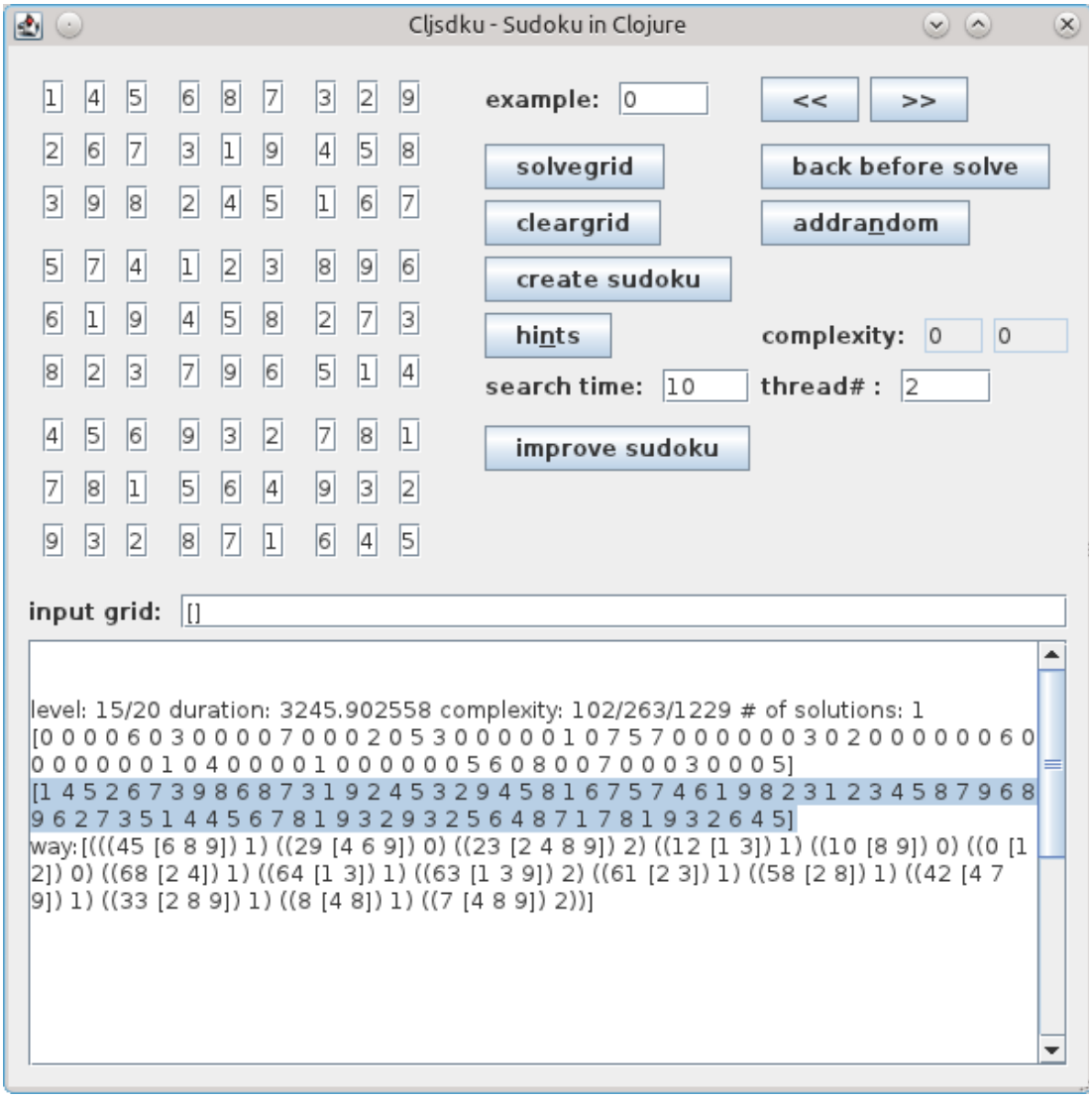


Illustration 1: The User Interface

2.1.1 General GUI Structure

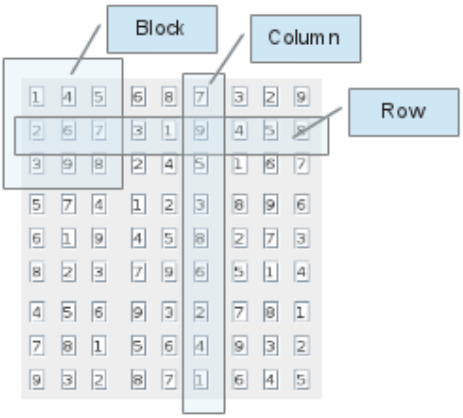
The userinterface is structured into the following three areas (Illustration 2):

Illustration 2: Raw Outline of the User Interface

- **Sudoku Grid**: contains 81 editable fields where the current puzzle is shown and can be changed.
- **Controller**: This area contains several buttons and input-fields usable to start and control the several usecases.
- **Output**: This is a scrollable multiline-textfield where several additional informations or errors are printed.

Besides the gui, some information also is output to stdout during calculations.

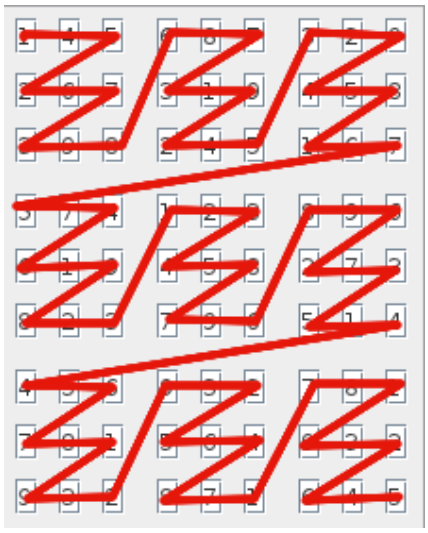
2.1.2 Grid Areas



- Block is one of the nine rectangular blocks
- Row is one row in the grid.
- Column is one column in the grid.

According to the sudoku-rules: the numbers 1-9 must be unique in each Block, Row and Column.

2.1.3 Mapping the Board to a clojure vector



Internally the sudoku-puzzle is represented by a linear clojure vector. The places are mapped in a Z-order to the vector.

The result of this mapping is:

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

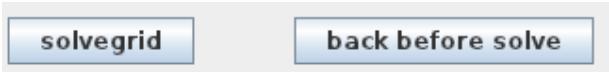
2.2 The Controlling Area

2.2.1 Scrolling through the examples

The Buttons "**<<**" and "**>>**" can be used to scroll through some example sudokus. The number of the current example is shown as well.

2.2.2 Solving puzzles

The Button "**solvegrid**" solves the currently shown puzzle, if possible. Using the button "**back before solve**" the solved puzzle can be reinstated.



When the solution could be created something like Illustration 3 is output. First some properties of the puzzle are output. The first level-number means how deep was the recursion necessary to solve the puzzle. The second level-number denotes how deep the maximal recursion was.

```
level: 15/20 duration: 3245.902558 complexity: 102/263/1229 # of solutions: 1
[0 0 0 0 6 0 3 0 0 0 0 0 7 0 0 0 2 0 5 3 0 0 0 0 0 0 1 0 7 5 7 0 0 0 0 0 0 0 3 0 2 0 0 0 0 0 0 0 6 0
0 0 0 0 0 0 1 0 4 0 0 0 0 1 0 0 0 0 0 0 5 6 0 8 0 0 7 0 0 0 3 0 0 0 5]
[1 4 5 2 6 7 3 9 8 6 8 7 3 1 9 2 4 5 3 2 9 4 5 8 1 6 7 5 7 4 6 1 9 8 2 3 1 2 3 4 5 8 7 9 6 8
9 6 2 7 3 5 1 4 4 5 6 7 8 1 9 3 2 9 3 2 5 6 4 8 7 1 7 8 1 9 3 2 6 4 5]
way:[((((45 [6 8 9]) 1) ((29 [4 6 9]) 0) ((23 [2 4 8 9]) 2) ((12 [1 3]) 1) ((10 [8 9]) 0) ((0 [1
2]) 0) ((68 [2 4]) 1) ((64 [1 3]) 1) ((63 [1 3 9]) 2) ((61 [2 3]) 1) ((58 [2 8]) 1) ((42 [4 7
9]) 1) ((33 [2 8 9]) 1) ((8 [4 8]) 1) ((7 [4 8 9]) 2))]
```

Illustration 3: Output after a grid could be solved successfully

Duration returns the number of milliseconds that where necessary for the calculation.

The following 3 complexity figures denote:

- Looking at the currently found way(described later) to the solution add the square of the number of the possible obvious decisions.
- Adds up the number of all superficially possibilities in all empty fields of the grid.
- The a sum of the squares of the numbers. So more possibilities at a specific place are valued more.

After that, the number of found solutions in solving this puzzle is given. Normally there is a limit of solutions the program calculates so this might not be a realistic number. But if the number is not 1 than the puzzle is no legitimate sudoku-puzzle.

The next line contains the original puzzle (a clojure-vector with 81 places) the line afterwards the solved puzzle.

If there is only 1 result, the way to the result is output. Each sublist denotes one successful step in solving the puzzle. The first number describes the place, the vector afterward the at that point possible entry-values and the number afterwards the index of the value successfully entered. The way does not contain single value results, they are something like spontaneous solutions and do not count refered to the complexity. Therefor all vectors in a way mus have more than one entry.

The currently best measurement for the complexity of a puzzle is calculated by squaring all the counts of vectors and adding them up afterwards. This measurement is used for the improvement of puzzles.

2.2.3 Creating new puzzles

Using "**cleargrid**" the grid is filled with zeros. Zero in a grid-position that the field is not filled out.

Using "**addrandom"** at a random empty position in the grid a random value is input. Before chosing a value a primitive check is maid whether the value does not oppose to other values in the current block, row or column.

Using "**create sudoku"** a sudoku-puzzle using the grid values is created. Possibly some of the original values are cleared or changed, possibly additional values can be found in previously empty places of the grid. This algorithm is quite fast but produces not very difficult puzzles.

Using the searchtime and the thread# input you can define how long the "**improve sudoku"** activity should at least last and how many threads should be used for the calculation. This improvement tries to create the most complex puzzle that can create the currently set grid-combination. For the best results the calculation should be based on a completely solved sudoku.

2.2.4 Getting Information about the puzzle

The "**hints"** button can be used to get help-information about the current puzzle. In the output something like picture 2 appears. First two complexity-values appear like described below. After that an ordered list describing the current possibilities is output. First the places with the least number of possible values appear. At the end the places with the most numbers. Each sublist contains first a description of the place (x/y(index in the internal array) last a vector containing all possible values. For example. the first entry in the picture depicts: At 1 based column 2 and row 4 only the value 3 can be input. Internally this place can be found in the array at the 0-based position 28.

```
complexity: 161/587
((2/4(28) [3]) (3/7(56) [3]) (3/2(5) [1 3]) (4/1(9) [1 3]) (2/6(34) [2 3]) (9/4(47) [3 4])
(9/5(50) [3 5]) (9/6(53) [3 5]) (3/8(59) [3 4]) (6/8(68) [1 3]) (5/9(70) [3 9]) (7/9(78)
[3 5]) (8/9(79) [3 5]) (2/1(1) [3 6 9]) (1/2(3) [1 3 6]) (1/3(6) [3 4 9]) (5/1(10) [1 3
4]) (4/2(12) [1 3 5]) (5/3(16) [3 4 5]) (7/1(18) [3 6 9]) (8/2(22) [3 5 6]) (9/2(23) [2 3
5]) (7/3(24) [3 5 9]) (8/3(25) [3 5 9]) (1/5(30) [1 2 3]) (3/6(35) [1 3 8]) (4/4(36) [1 3
9]) (6/4(38) [1 3 8]) (6/6(44) [1 3 8]) (1/7(54) [2 3 6]) (2/7(55) [2 3 6]) (1/9(60) [3 8
9]) (5/7(64) [3 6 7]) (7/8(75) [2 3 5]) (5/2(13) [1 3 5 8]) (8/1(19) [3 6 7 9]) (5/4(37)
[1 3 8 9]) (4/5(39) [1 3 5 6]) (8/5(49) [1 3 5 6]) (7/6(51) [3 5 6 9]) (4/8(66) [1 3 6
9]) (9/8(77) [2 3 5 7]) (1/1(0) [1 3 4 6 9]) (1/6(33) [1 2 3 7 8]) (5/5(40) [1 2 3 5 6])
(8/6(52) [1 3 5 6 9]) (1/8(57) [2 3 4 6 9]) (2/8(58) [2 3 5 6 9]) (5/8(67) [1 3 6 7 9])
(5/6(43) [1 2 3 5 6 8 9]))
```

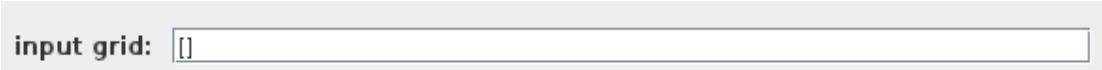
Illustration 4: Output after pressing the hints button

The third sublist describes the free place in the third column and second row. Which internally can be found at index 56 in the array. You can only input the values 1 or 3 there.

The two complexity-fields return two complexity-measurements of the current sudoku. The first adds up the number of all superficially possibilities in all empty fields of the grid. The second is the a sum of the squares of the numbers. So more possibilities at a specific place are valued more.

2.2.5 Fast Input of Puzzles

Often puzzles or solutions or intermediaries are output either in the console-output or the GUI-Output-Area. The format is the clojure vector format. An easy way to input such a puzzle is by pasting it into the **"input grid"** - area.



3 Algorithms

3.1 Basic structures

3.1.1 Sudoku board

The sudoku-board is internally represented by a clojure vector mapped in Z-Order to the real board. (see Chapter 3)

3.1.2 Possibilities List (pbls)

Given a sudoku-board with empty places, derived from the already filled out neighbours of the block, row and column where an empty place is located, a list is generated containing the information which values might be inserted.

An example:

> ((46 [9]) (22 [2 9]) (49 [6 9]) (51 [3 6]) (53 [3 5]) (68 [1 3]) (69 [1 4]) (77 [1 8]) (78 [1 9]) (7 [1 3 8]) (8 [1 8 9]))
>
> means:

Position	Values which may be inserted	Grid Location (column/row)
46	9	8/4
22	2, 9	8/2
49	6, 9	8/5
51	3, 6	7/6

Position	Values which may be inserted	Grid Location (column/row)
53	3, 5	9/6
68	1, 3	6/8
77	1, 4	4/9
78	1, 8	9/8
7	1, 3, 8	7/9
8	1, 8, 9	2/3

In the GUI-Output, this list looks as following:

((8/4(46) [9]) (8/2(22) [2 9]) (8/5(49) [6 9]) (7/6(51) [3 6]) (9/6(53) [3 5]) (6/8(68) [1 3]) (4/9(69) [1 4]) (9/8(77) [1 8]) (7/9(78) [1 9]) (2/3(7) [1 3 8]) (3/3(8) [1 8 9])

The list is sorted so, that the sub-list containing the shortest value arrays are at the beginning. Single value entries are handled in a special way since they unconditionally define that an entry may be set without necessary recursion.

3.1.3 The “Ways List”

In a normally solved sudoku-puzzle only one way is returned here. The way does not contain the spontaneous filling out caused by single value entries of the “Possibilities List”

An example:

[(((8 [1 9]) 1) ((7 [3 8]) 1) ((78 [1 9]) 1) ((68 [1 3]) 0))]

Position	Values which may be inserted	Index of chosen value	Chosen Value
8	1, 9	1	9
7	3, 8	1	8
78	1, 9	1	9
68	1, 3	0	1

The “Ways List” has a string relationship to the “Possibilities List” but the value-vectors may be subsets of the vectors there because they can become smaller during the filling out of neighboring (block, row, column) places.

3.2 Solver

The solver is implemented as a simple backtracking algorithm.

1. A “Possibilities List” is created.
2. As long as single value vectors exist in the list, these entries are made in the board. The “Possibilities List” is always adapted accordingly. The possible outcome of this could be, that a complete board is solved without having to do a single recursion.
    - If during the adaption of the “Possibilities List” a values-vector becomes empty, the current path is not solvable, a backtrack if possible is necessary.
3. Now the “Possibilities List” contains no single value elements any more (possibly she never did). One of the entries with the shortest value-vector is selected and the first value is chosen to be filled in. After adaption of the “Possibilities List” accordingly new single-value entries might have been created. Back to step 2.
    - If the “Possibilities List” does not contain any entries anymore, a solution has been found. Nevertheless if the number of solutions is not bigger than a given value the algorithm does a backtracking here. It must try to create a second solution to make sure, that the uniqueness-constraint of the puzzle is not broken.
4. Backtracking: If a step 3 comes back without solution, the next value out of the current values-vector is chosen. Back to step 2
    1. If no values-vector entry is left, the chosen path can not be followed anymore, further backtracking is necessary.

The solver generates a result (clojure structure `cljsdku.solver/solveres)` containing the following information:

- **:found?** True if at least one solution could be found.
- **:result** The last result found
- **:level** The recursion level where the last solution has been found
- **:results** All the results up to a maximum number defined internally or by parameter.
- **:ways** The possibly different ways leading to the different results. Used in the first generation-algorithm to make sudoku-puzzles leading to unique solutions. Used in the second generation algorithm to evaluate the difficulty of an array.
- **:maxlevel** The maximum recursion level necessary during the creation of a solution (and making sure that it is unique)

3.3 The first sudoku generator

This Generator tries to create a sudoku-puzzle with a single solution based on a solving result.

1. It tries to solve the puzzle
    1. If no solution exists, it removes an arbitrary value out from the board, back to step 1,
    2. If exactly one solution exists, it returns the current puzzle where the solution was based upon.
    3. If more than one solution exists comparing two of the returned ways some places are filled out, to make sure that the degree of freedom is limited at critical points. Back to step 1.

3.4 The second sudoku generator (improver)

Basis of this algorithm is an at least once solvable puzzle or a complete board. Now values are cleared from the board and checked whether anyhow a single solution can be generated.

4 Structure

5 Installing