Project Report

Edon Kelmendi

November 20, 2011

1 Introduction

Sudoku is a combinatoral number placement puzzle. It is usually played in a 9×9 grid. That is a grid which contains 9 columns and 9 rows, and whose elements are called cells. Grids also contain 9 blocks which are smaller grids of size 3×3 that do not overlap with each other.

Some of the cells might be filled with numbers from 1 to 9 and others might not be. The ones that are already filled are called hints.

The purpose of the game is to fill the unfilled cells with numbers from 1 to 9 such that no two numbers are repeated in the same row, column or block. Finding a solution to this puzzle is known to be an *NP* problem.

In this project the size of the grid to be solved/generated is generalized but a constant is used as a higher bound (by default set to 64).

2 Heuristics

2.1 Locked candidates

One of the heuristics implemented on this project besides the cross-hatching and lone number one, is the one called *removal of the locked candidates.*

Since we have the requirement that the numbers inside a block should not repeat this heuristic looks for a number in a block which is to be seen only on one row (column) and removes it from the rest of the row (column). Since no matter the state of the grid at later points in time that certain element is going to be in that certain block at that certain line (column).

This was done using two functions **rm_locked_candidates** which takes as one of the parameters the number of the block on which we want to look (so it must be run over all blocks), and senses if there is a candidate locked in a row or column that, of course, is not a singleton. If it finds one (or more) it calls the **cross_off_candidate** function to remove the locked candidates from the rest of the row or column.

Therefore the job of **cross_off_candidate** is to remove some set of colors from the cells in the row (column) but not from the certain block.

This heuristic makes the program less efficient in smaller grids but, with it the program performs better in bigger grids. And since smaller grids are solved relatively fast anyway, in general, such a heuristic seems to be helpful.

2.2 Naked set

The other heuristic implemented is the *naked set* one. The gist of this heuristic is this observation: if we can find in a subgrid n colors of cardinality n which are all equal to one another then it is safe to cancel all the elements of such a set from other cells of a subgrid.

A subgrid is a row, column, or block.

This heuristic is implemented in the function **naked_set**, which in contrast to the locked candidates heuristic can be mapped over all the subgrids of a grid, same as with other heuristic functions. This functions helper is **rm_naked_set** which cancels the colors of a naked set from a given subgrid while leaving the ones that are first encountered.

3 Grid Generation

Generating a grid is harder, computationaly, since (at least in this case) it involves solving the grid a number of times.

3.1 Unstrict grid

Generating an unstrict grid of a certain size involves solving an empty grid of that size, and at points where we have to make guesses, the guesses that we make should be random. For performance reasons we choose by random from the cells with smallest cardinality.

This way we get a new randomly generated but already solved grid. Then we just take out 2/3 of the cells. This, of course, does not guarentee that there will be only one solution, but just that it will be at least one solution.

3.2 Strict grid

We do the same as with the unstrict grid, up to the point of cleaning off the 2/3 of the cells, because now we need a unique solution.

We could have just started removing cells one by one, and always checking if the grid is solvable using only heuristics, and if no we would stop. But this doesn't mean that we can't remove more numbers from cells and still have a unique solution. And such a thing would lead to boring puzzles with lots of hints.

So the road that we take is this: we define a function **num_of_solutions**, which does what it name implies, it returns the number of solutions that a given grid has. And then generating a strict grid is reduced to removing numbers from cells randomly and calling this function on the resulting grids to see if the return value has changed from 1. When it does, the step before hand will be the generated strict grid.

Note that finding lowest number of hints required such that the grid has a unique solution is still an open problem. This algorithm will just generate grids with different numbers of hints.

It also should be noted that in this specific implementation the seed value for random number generation is taken as the number of seconds past the epoch. So one can't generate two different grids in one second, they will be the same.

4 Implementation

The whole program is divided into five modules:

1. **main**: Holds the main function and closely related functions to parsing command line parameters such as **usage** etc.
2. **parser**: Holds the parser for the grids that are given as input and other helper functions, mostly for handling parser errors.
3. **preemptive_set**: Implementation of preemptive set data structure, in our case a 64 bit bitfield but such a fact is well hidden by the interface, which exports all the neccessary operations. Can be used as a stand-alone library.
4. **heuristics**: Implements the above mentioned heuristics for solving the sudoku puzzles, together with a set of a much needed helper functions.
5. **sudoku**: Holds both the sudoku grid solver algorithm and the generation one, has quite loose coupling with the heuristics module just calls one function defined in it.

Optimizations taken were simple ones, such as checking if a grid is consistent without doing too many steps with heuristics saved a bit of time.

Tried making the above mentioned modules as loosely coupled as possible, in such a way that there is a very small interface between them and in the future the parser or euristic module can be changed efficiently.

5 Code quality

Testing the program on lots of grids was chosen as a way to ensure the quality and robustness of the program. This task was relieved by the generation part (we could now generate new test cases), and a huge set of unsolved nine by nine grids. On most of these tests (as long as it was practically feasible) valgrind was run to ensure that memory leaks are not possible.

Testing on grids of size 9×9 was more practical because if we would see an unwanted behaviour it would be easier to track where the program went wrong just because of the size of the grid. While on smaller grids the possibility of problems arising was smaller and on larger grids it was too hard to see where the problem is arising.

Another reason why the test were usually made on $9x9$ grids is that we had a big set of grids which we knew to be solvable taken from:

http://mapleta.maths.uwa.edu.au/ gordon/sudokumin.php
.These are 49151 distinct sudoku 9x9 grids with 17 hints.

The format on which this set of grids is given can't be parsed by our parser module, but they can easily be turned to one a job which the script **pandg.rb** does (careful when you run it, it will spam your working directory with 49151 files named *grid-n* where n is the n-th grid. One can then run the program on these generated files.

Other measures were taken on ensuring the robustness of the program such as trying to make functions less depended on one another or trying to make this dependence more explicit, qualifying variables with const when we're sure that it won't be changed, and other commonly used rules that help.