Programmation C: Sudoku

LOUISO Hugo

24 Novembre 2023

Contents

1 A word from me about this project

After a Fundamental Mathematics bachelor, I never programmed in C language. All the programming background I had was Python coding, with manipulating objects as lists, arrays, and a lot of mathematical objects. This project made me realise that Python is a really, really easy and malleable language where the programmer is such an assisted. For example, one of the most basic thing we learn in Python is to reshape a list. In C, I discovered the notion of Heap and Stack and learned that only the Heap objects (malloc, realloc) can be reshaped. I figure it would be shocking for an exclusively C programmer to hear that Python functions can even not have a specified type as entry and sometimes return a string, sometimes an integer or a boolean... It was definitely very instructive to learn a low-level language for the first time.

Furthermore, I discovered an essential notion of C: pointers. In Python, I used to manipulate objects by always overwriting them with function returns. I discovered that it could be really more effective to use pointers in functions.

2 Heuristics

I decided that my function grid_heuristics() would call the less complex heuristic as many time as they no more have effect to call the next heuristic (with a complexity "one-step" higher). The reason is I think we need to benefit and gain information from less complex functions as much as possible.

2.1 Heuristics that I implemented

2.1.1 Cross-hatching

Cross-hatching is the most obvious heuristic possible. You just visit every subgrid of your grid and discard every singleton from other cells. That's a basic reasoning that all novice players of sudoku do.

My implementation is: for each subgrid, creating a color and give it all colors that are singletons in the subgrid. Finally, just discard this color from all cells that are not singletons.

Complexity: $\mathcal{O}(n)$ (n = grid/subgrid size)

2.1.2 Lone number

Lone number is a bit harder for a human being. We just look for all cells in a subgrid and if a color appears only once, we know its cell can actually only be this color.

For each subgrid:

- I create an array of the same size as the subgrid. Each cell of the array represents the occurence number of a color.

- I visit every cell of the subgrid and increment the array cell of the color if I find the color in the cell of the subgrid.

- Now, I just have to look which cell in the array has 1 as value, and find the cell in the grid that has the corresponding color, to discard all others from this cell.

Complexity: $\mathcal{O}(n^2)$ (n = grid/subgrid size)

2.1.3 Naked-subset

Naked-subset aims at searching for N cells that are equal with N colors. And we know we can discard this color from all other cells. Let me explain you how I implemented it, for each cell in a subgrid:

If the cell is a singleton I obviously do nothing, else, I count how many other cells are equal to this cell, if the response is the number of color in the cell, I won, and can remove these colors from all the other cells.

Complexity: $\mathcal{O}(n^2)$ (n = grid/subgrid size)

2.2 Hidden-subset?

2.2.1 What subset

I coded a last heuristics: hidden-subset. It could be a very complex implementation if we want it to be "complete", let me explain.

First, we need to take a set of N colours that are present in exactly N cells. What I mean by present is that there is exactly N cells in the subgrid which intersection with the set is non-empty. The thing is, we have a multitude of ways of choosing the set of N colours. We can build it with unions of 2 colors from the subgrid cells, union of 3, 4... If grid size is n, we have $\binom{n}{k}$ choices to make a union of k cells. This can quickly be a huge number for a great size n of grids (the ones which interest us if I'm right).

Here I'm comparing $\binom{n}{2}$ with $\binom{n}{3}$ for n = 36, 49, 64 (hugest sizes of grids)

n	36	49	64
$\binom{n}{3}$ / $\binom{n}{2}$	≈ 11.7	≈ 15.7	≈ 20.7

What this array means, is for example that in size 64, hidden-subset will be around 22 (21 + 1) times longer to get applied if we choose to check all unions of 2 and of 3, instead of only unions of 2.

That's why I decided to stick with k = 2 cells, and not making unions but intersections. I chose intersections because I figure I have more chance that this heuristic works if I choose a set with a small size N. I would have liked to compare time it takes to solve a lot a of grids choosing 2-unions instead of 2-intersections, but I'm short of time and I prefer developing the two next aspects of my work: solver and generator.

2.2.2 Am I using hidden-subset

Moreover, I realized that my solver could sometimes be really quicker when I don't use hidden-subset.

For example, for solving the 3 grids of size 25 of your directory "grid-solver" with my hidden-subset, my solver takes 2m56. But without it takes only 24,4s. For above sizes, the comparative data was just too random, but notably, my generator didn't work anymore in size 64. So I decided to just never use hidden-subset.

3 The backtracking solver

I actually have 3 functions called backtrack:

"backtrack_first" searching for the first solution to a grid.

"backtrack_all" searching for all solutions to a grid.

"backtrack_unique_solution" searching for all solutions but stopping if finding two.

All use a recursive method to find solutions.

Moreover, all are void function, because I decided to work only with pointers, and actually with the same and unique pointer all along the function execution.

Because they are obviously very similar, I will describe only the implementation of backtrack_first:

My function backtrack_first take a pointer to a grid and a pointer to a Boolean "solution_found" initialised at false. (If you looked at my code, don't pay attention to the Boolean "random" in entry, which is for generator use.)

- First thing I do is applying my heuristics on my grid.

- If the result of it is "inconsistent": I simply stop the function with a "return;". If it's "grid is solved", I set the Boolean "solution_found" at true and stop with "return;".

- If both didn't happen, it means the grid is just not solved, so we have to make a choice. That's what I do after having created a grid_t *copy of my grid. Now, I call backtrack_first on my grid (not on the copy).

- After this backtrack_first call has finished, I need to look if a solution has been found by looking at the value of the Boolean. If yes, I stop the function, else I discard the choice I made earlier from the copy, and overwrite grid with this choice discarded copy.
Now I have to call backtrack_first on my choice discarded grid.

4 Generator and some probabilities

4.1 My generator

Generated grids need to be random, so we don't always generate the same grids. This is how I built my generator:

- I create a totally empty grid, with a colors_full(size) in every single cell.

- I run backtrack_first on it but with the "random" Boolean set as true. The effect it will have is that when making a choice in the backtrack function, it will not call grid_choice which as you ask selects the rightmost color. Instead it calls grid_choice_random which selects a random color in the cell where we make a choice.

- Once I have a solved grid, I hide a cell in the grid with a probability of 1/4.

- If the user ask for a grid with a unique solution, I need to check if there is at least 2 solution, by calling the Boolean function "solution_is_unique" which one calls "backtrack_unique_solution". If that's the case, I need to recursively call back my function to try again.

4.2 Chance

This method led to a very counter-intuitive fact: my solver could take several minutes to solve grids of size 25, and just cannot solve some grids of size 49, 64. But, with my generator method, it can sometimes solve a totally empty (with absolutely zero information) grid of size 64 in around 20 seconds ? That's just startling, because we could think that a grid of size 64 that already has some filled cells is easier to solve because my heuristics will benefit from this dear information.

The thing is, my random resolution can sometime make random choices that lead to a grid that is easy to solve (doesn't need a lot of backtracked choices), and sometimes to grid that will need to make a very lot of choices and backtracking to be solved.

However I have to say I was a bit surprised that I often succeed on generating a grid of greatest sizes under a minute, and generate a 36 sized in 1.2 s almost all the time. I think that's a surprising distribution of chance which we could search for the reasons. But again for a lack of time, I won't search for these, I prefer finishing my work by some statistics of success of my generator which stays in theme of this section.

For sizes 36, 49 and 64, I generated several grids with time option.

For size 36, the values I got (in seconds, with a CREMI machine) for generating a grid of size 36 were:

1.265, 1.214, 1.232, 1.216, 1.249, 1.249, 1.242, 1.251, 1.245, 1.219, 1.238, 1.272, 1.225, 1.286, 1.242, 1.277, 1.246, 1.243, 1.233, 1.220.

It gives a standard deviation of around 0.020, which is very nice.

Moreover, my generator never blocked for too long (more than several seconds) when generating a grid of size 36, which make me conclude that it is working for grid under 36 sized. However as there is some grids of size 36 that I can't solve, I know it's still possible that it blocks.

For size 49:

I tried to generate several grids, I the durations I got were:

5.574, 5.378, 5.667, 5.402, 5.581, but when I tried to generate another my software just didn't stop, and I stop it after 10 minutes.

For size 64, it seems that around half of the time it generates a grid in around 20 seconds, and the rest of the time in several minutes or just "never".

5 Limits of my work

Even if I already said about it, I will give the points I would have liked to search a bit more, if I had the time.

- Maybe the complexity of my heuristics could be better.
- There was a lot of probability aspects in my grid generation and I don't know if these were hard to explain.
- I'm not solving a lot of challenges grids.
- Maybe I could implement others heuristics, but all I found were quite complex.