TAKUZU PROJECT

Assignment 6: report

Author
Briac Bruneau
M1 Crypto
décembre 2023

Contents

1 Introduction

First, I need to explain how I decided to represent my grids because it has a rather big impact on how i approached the heuristics, the choices and the generation of grids. Instead of using pointer of characters or pointer of pointers of characters i decided to represent my grids as a pointer to tabs of `uint64_t`, each of the `uint64_t` representing the positions of the zeros and ones on a grid. This representation makes the code look heavier but it makes comparisons and data access way faster because of bit by bit operations, and the objects are way lighter. I also decided to use lines AND columns, not because we need it but because it makes heuristics faster aswell.

I also want to say that everything i say in this report is non-exhaustive and that the vast majority of choices I made in the implementation of this takuzu solver/generator is explained in comments in the code. In the following parts, n will be the size of the grid and N the percentage of filling of generated grids.

Listing 1: Grid Structure

```
typedef struct
{
  int size;
  binline *lines;
  binline *columns;
  int onHeap;
} t_grid;
```

Listing 2: line structure

```
typedef uint64_t binline[2];
```

2 Heuristics

2.1 Consecutive cells heuristic

This one is pretty straightforward, following the rules of takuzu, if there are two consecutives 1 (resp. 0) then we put a 0 (resp. 1) before and after, thought my implementation can appear scary at first sight. Basically I use bit by bit with a line to isolate bits where we have a two times the same character consecutively.

Here in Example 1, let's consider it represent the position of ones, we can see bits 1 and 5 are present meaning we have 2 times the same character consecutively on position 1/2 and 5/6, we then can put zeros on position 3, 4 and 7, if these cells are empty. I admit it is a bit confusing in the beginning to understand how is represented my grid but keep in mind i have one integer for the position of ones and one integer for the position of zeros, all this for each and every line!

Example 1:

$$\begin{array}{r} 10110011 \\ \&01011001 \\ \hline 00010001 \end{array}$$

For my implementation, I decided to write a function `consec_subheuristic` applying this rule to the line it takes as an argument, and my function `consecutive_cells_heuristic` call the subheuristic on every line and column.

2.2 Half line heuristic

I also divided this heuristic into a subheuristic `half_line_filled` that I call n times. The subheuristic uses `gridline_count` to count the number of ones and zeros in the binary of line i and if this count is higher than half the size of the grid, fills the rest with opposite characters, same for column i.

So if we take a look at example one again, instead of looking at consecutive bits, we can just fill the rest of the line with zeros because we can observe 4 bits activated on the integer representing the position of ones : half the line is filled with ones.

2.3 Inbetween cells heuristic

Same as the 2 others, the main heuristic calls `inbetween_subheuristic` on every line and column, and with almost the same operation bit by bit than consecutive cells heuristic we this time bit shift 2 times to isolate bits where we have the subset : '101'. Thus meaning the bit in the middle must be of the opposite type else we would have 3 times the same character consecutively on one line, ending in an unconsistent grid.

2.4 Optimisation

Representing the grid as tabs of 2 binaries allows me to bypass going through the tab and drastically reduce complexity since i only need one operation PER line for EVERY heuristic! I still need to look into the isolated bits to fill the grid but in the worst case i end up with a time complexity in O(nlog(n))

I also decided to run throught the first heuristic until it doesn't change anything in the grid anymore before using the 2nd one and same for the 3rd one because the consecutive cells heuristic is more efficient in the beginning.

3 Grid Generation

For the whole grid generation section, small grids will be grids of size : 4x4, 8x8, 16x16, while big grids will be grids of size: 32x32 and 64x64.

3.1 1st sketch

In the assignment 4, we had to code a basic generator containing N% of O and 1, it didn't need to have a solution. So i just initialized a tab of size n*n with indexes from 1 to n*n and suffled it, then placed 0's and 1's to the first indexes of the shuffled tab until it was filled enough.

This generator is fast but doesn't always generate a solvable grid... from then on, even tho filling a grid until it has the good percentage is possible, I only worked on generating a consistent grid and removing cells from it.

3.2 2nd sketch

I then tried to generate a grid with backtrack. First I filled 25% of the grid with random placed 1's and 0's and then i called my grid_solver on it to find a solution. It worked well for small grids, even thought it sometimes struggled with 16x16, but it took forever to find a solution for big grids... I let it ran for about 10 minutes and it didn't find any solution for a 32x32 grid, the number of possible grids is simply too big. I needed to use a different method.

3.3 final sketch

I finally found a working solution : generate 4 grids with half the size, representing the 4 quarters of the grid we want to generate: top left, top right, bottom left, bottom right. And then assembling them together.

My first thought with this approach was that I needed to control the outside ring of the grids in order to not have 3 times the same character in a row when assembling them to each other as show in example 2.

Example 2:

$$\begin{bmatrix} \_ & 1 & 0 & 0 \\ \_ & \_ & 1 & 0 \\ 1 & \_ & 0 & 1 \\ \_ & 0 & \_ & \_ \end{bmatrix} \xrightarrow{\text{concatenate}} \begin{bmatrix} 0 & \_ & \_ & \_ \\ \_ & \_ & 0 & 1 \\ \_ & 1 & \_ & 0 \\ \_ & \_ & 1 & \_ \end{bmatrix}$$

Example 3:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

to resolve this issue, all corners of the subgrids were as in example 3 to avoid any concatenating conflicts. Then i proceeded to fill the 2 outer lines of each subgrids with one being the opposite of the other: the goal ? never having 2 times the same character when going from the outside ring to the inside. Because it is easier to vizualize, my `grid_outer_ring` function generates something like example 4.

Example 4:

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & \_ & \_ & \_ & \_ & 0 & 1 \\ 0 & 1 & \_ & \_ & \_ & \_ & 0 & 1 \\ 0 & 1 & \_ & \_ & \_ & \_ & 1 & 0 \\ 1 & 0 & \_ & \_ & \_ & \_ & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

This generation made the assembling super easy and I encountered no difficulty in finding a consistent grid this way (simply called `grid_solver`).

Of course adding up 4 subgrids together won't always result in a consistent one but I can just repeat the process, it's very efficient.

3.4 Optimisation

In the end, for small grids, I had one generator which generates the outer ring and another which fills 25% of the grid and then both of the call `grid_solver` to find a solution. And for big grids I only used assembling method because the 2 others weren't working at all (or I didn't have the patience ? We'll probably never know).

I compared the time it took to generate small grids between the 2nd and 3rd generator (no assembling, just outer ring + backtrack). For 4x4 and 8x8 grids there was barely any difference, but once we test on 16x16 grids, the 3rd generator showed to be much faster, notably because the 2nd one sometimes ($\tilde{2}$0% of grids) encountered a wall and took a few seconds to generate, as opposed to hundredths of seconds usually.

So in order to upgrade my generator, I decided to discard the 2nd generator and to keep the 3rd one only. I used the generator for grids of size 4 (faster than assemble because it's essentially the same code but with less conditions and no concatenating). For grids of size 8x8 and + I used `grid_assemble` which recursively creates 4 grids with half the size with assemble until it reaches size 8 where it calls `grid_generate` to form 4 grids of size 4x4 and assembles them.

When it comes to emptying the grid, I use the same system of random indexes and empty the grid until we reach the threshold. For unique generation, I also use the random indexes, I copy the grid and remove the cell to copy, then call the solver on copy with MODE_ALL and if more than one solution is found i skip to the next index. if the grid isn't emptied enough, I call `grid_remove_cells` again, it generates a new shuffled index tab and does the same thing. If after 10 iteration we haven't found a emptied grid with unique solution, I generate a new grid because i consider emptying this one in unique mode isn't possible. With this method, it is only possible to generate grids up to 16x16, I haven't found any way to generate any bigger grid with unique solution.

For all generation I used N = 0.3, but we can change it as we want since the generation time doesn't depend on N but on the grid size. (this applies only to non-unique generation) s

4 Conclusion

I introduced some macros to make the code easier to read and understand and made sure to free pointers because i had some memory leaks. I need to allocate dynamically my grids in generator and copy in solver because the function can return those and it can't return a pointer to a locally allocated variable.

I also put a DEBUG rule in the Makefile to be able to used gdb because i had issues locating one of my memory leaks. I also used a lot the operation : ¡ int + '0' ¿ to transform int into a char. I also modified my choice method in order for it to chose more wisely : it looks at the line with the least empty cells and randomly fill the most isolated one. Finally, I made it so that verbose shows you how many backtracks were necessary, shows every choice made and shows time elapsed in generation (pointless in solving since printing choices make it way way slower).