Takuzu - Report

BOUCHARD Corentin

December 13, 2023

1 Introduction

This is the report explaining my work on the Takuzu project. I've worked alone for all of the project except for some useful dicussions with Briac Bruneau and Tony Law. This report will mainly go through : heuristics and grid generation. The generation of grid with a unique solution have not been implemented.

2 Heuristics implemented

Only very basics heuristics have been implemented. Many more could be implemented but we would have to check if they improve the solver. After looking at which heuristics are used we will explain how my backtracking works.

2.1 heur_consecutive

Using the rule that a well-formed takuzu cannot have 3 consecutive same characters, this heuristics go through all the grid to find 2-consecutive characters. Whenever found, it tries to update empty cells on the top and bottom (or left and right for lines) to the other character.

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

2.2 heur_fill

Takuzu grids must have exactly half 0 and 1 on each line/column. Whenever a line/column already has half its size filled with one characted, the rest can be filled with the other character.

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

2.3 heur_between

Whenever we have a 1 (resp 0) a blank then an other 1 (resp 0), we know that the blank in the middle must be a 0 (resp 1). Otherwise we would have three consecutive characters resulting in an inconsistent grid.

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

2.4 Backtracking

I have cut backtracking in two functions, one finding only one solution (the "first" one), one finding every solutions. Both methods requires that we first apply heuristics before entering backtracking functions. The choice function is the same for both, it finds the first empty cell, starting from top left. This is probably a bit better than choosing random, because after many calls, choosing from the same area will result in heuristics filling more cells. If more cells are filled, we know quicker if the current grid is going to be valid or not. The current backtracking takes too much time to solve grids of size 32 or 64.

2.4.1 Finding first solution

Entering the function, we have a non-full, non-improvable via heuristics, consistent grid. We make an arbitrary choice in the grid and reapply heuristics. If the grid is valid (consistent and full), we have found a solution and returns it. If the grid is not consistent, we change the choice made (change the character) and reapply heuristics. We return either the grid or we keep exploring depending on the validity. We then explore further the grid, and at the end of the recursive stack, we check for consistency, and change the choice made if we ended up on an inconsistent grid.

2.4.2 Finding all solutions

Everytime the function is called, we create a two copies of the grid. We make an arbitrary choice for copy1, and we change the character for copy2. We apply heuristics on both copy and we print the solution, continue exploring, or stop, depending of the validity of the grids.

3 Grid generation

The grid are generated randomly, we make sure they are solvable (for size ¡ 32) by solving them using the backtracking. The generation for grids of size 32 or 64 often leads to a consistent grid, but inconsistent after applying heuristics using this method.

3.1 generate_rand_grid

We start with an empty grid of a given size. We have to fill N% cells of the grid, we generate a random position on an empty cell and fill it with a random character (0 or 1). We check if the grid is not consistent we try the other character, and if it's still not consistent, we start over from an empty grid. We repeat the operation until the grid is filled with N% characters.

3.2 generate_solvable_grid

We use the last function to generate a random grid. If the size of the grid is less than 32, we solve it using the backtracking. If the grid is solvable, we can return the grid, otherwise we repeat the operation until we get a solvable grid.

4 Final improvements

This section will explain some of what happens since assignment-5.

4.1 Tester

I've greatly improved my tester script done for assignment-4. Added many grids, to different folders depending on if they are inconsistent, if they cannot be parsed, if they are solvable or not. The tester now checks if it is executed from the right folder and if make have been called before. Also added testing of valgrind on different arguments. All these tests made me found small bugs and memory leaks and they were also very useful to check if everything was still working after making small improvements.

4.2 Cleaning, improvements

Some documentation have been added, and some have been moved to header files. Most improvements come from memory leaks, every memory leak found with valgrind have been fixed. A target have been added to make the pdf of the report.