Takuzu

Assignment 5: Backtracking

The objective of this assignment is to develop a first version of the complete project.

You will work alone on this assignment.

As you may have noticed in the last assignments, heuristics may not be sufficient to completely solve a grid. We will then proceed as follows.

- We start from the initial grid and we apply the heuristics.
- Once the heuristics have exhausted their guesses, we perform an arbitrary choice over the grid and re-apply the heuristic. If the heuristics did not guess a solution again, we do another arbitrary choice, and so on until we reach a leaf of the search tree.
- At the end of the exploration, we will either reach a solution, or have an invalid grid. In the latter case, we need to backtrack to the last choice we made, discard the wrong choice and try another choice.

Grid copy will be used to manage backtracking: before making a choice, the grid will be copied (call to function `grid_copy`) and the exploration will operate on the copied grid.

1. Choice method

The choice of cell values to be explored need not be totally random, we can choose a cell that will give solutions with a higher probability. The objective of this exercise is to set-up the data types and the functions that will help you to make smart choices.

1. In the grid module, add a type that represents the choices:

```
typedef struct {
 int row;
 int column;
 char choice;
} choice_t;
```

   Choices in the structure may contain one bit per choice (ZERO or ONE)
2. Write function `void grid_choice_apply(grid_t *grid, const choice_t choice)` that applies the unique choice in `choice` to a given grid
3. Write function `void grid_choice_print (const choice_t choice, FILE *fd)` that prints the choice made on a given file. This function will be used for debugging your code.
4. Write function `choice_t grid_choice(grid_t *grid)` that selects the best choice to do for a given grid. In a first attempt the choice can be made randomly if you don't find a better solution.

2. Backtracking

1. Define and fill-in, using the command-line parameters, a structure of type mode_t, that defines if we want all solutions or only the first one.
   typedef enum { MODE_FIRST, MODE_ALL } mode_t;
2. Write a grid solving function: `grid_t *grid_solver(grid_t *grid, const mode_t mode)` that applies the heuristics and makes arbitrary choices but without trying to backtrack. The result of such a function can be either a solved grid or an inconsistent one. Debug messages will print the choices made, the result of the exploration and the output grid.
3. Implement backtracking and the code to find the first solution only (mode `MODE_FIRST`) and print the solution It is recommended that you used a recursive approach to implement backtracking but the choice is yours.
4. Add the code to search for all solutions. In this case function `grid_solver()` may return NULL, all solutions will be printed, as well as the number of solutions found. In order to ease automated testing, the output of the software (when using option `'o'`) will the following:

```
Number of solutions: X (zero if the grid has no solution)
Solution 1
Grid for solution 1
…
Solution X
Grid for solution X
```

3. Grid generation

Modify you grid generator such that it is capable to produce grids that have at least one solution.