Advanced Programming

Solving Sudokus

1 Context

Sudoku is a popular game that is played on a 9× 9 grid, some cells containing digits between 1 and 9 and some empty. Playing the game amounts to filling the grid with only 4 rules:

- Each cell should have a digit between 1 and 9
- Each row should contain all digits
- Each column should contain all digits
- Each of the 9 sub-square of dimension 3×3 should contain all digits

The difficulty of a Sudoku represents how many candidates exist in each empty cell at the beginning. An easy Sudoku will contain some cells having only one candidate, leading to a sure guess. A difficult Sudoku will contain only cells having several candidates, forcing the player to rely on hypotheses until a contradiction occurs.

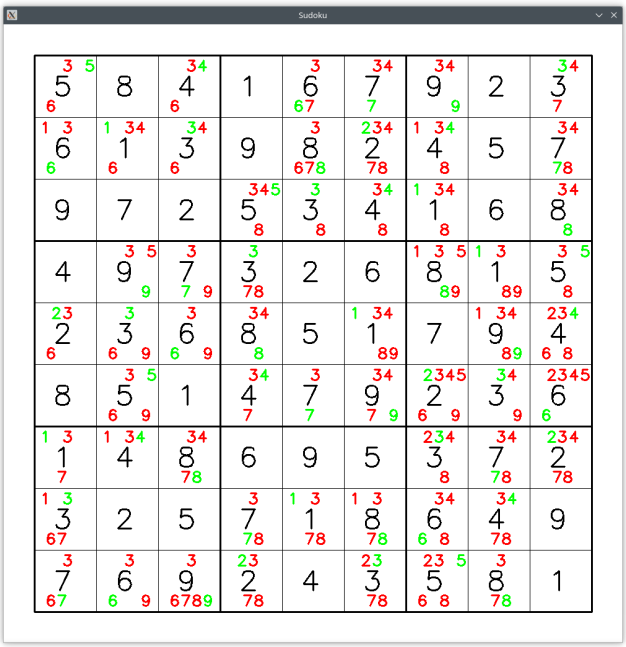

Figure 1: A solved Sudoku. Red digits show initial candidates for each cell. Green digits are the initial candidate that was the valid one at the end. Cells without red/green digits are the ones from the starting grid.

2 Backtracking algorithm

Backtracking is a simple algorithm that can solve Sudoku quite well. It solves the cells one at the time, possibly doing a guess among the candidate digits. If it finds a contradiction, is cancels the previous guess and tries the next one. For an easy Sudoku, this algorithm will never have to do any wild guess and will solve the grid without ever cancelling a guess.

The algorithm reads as follows, under the `solveNextCell()` function.
This function is recursive as once a guess is done on a cell, it calls `solveNextCell()` again.

```
Function solveNextCell()
if grid is full then
    return true
end
next_cell ← best next cell to investigate
for guess in next_cell.candidates do
    // let's assume this guess is the correct one
    Assign guess to next_cell // also prune from neighboors
    if solveNextCell() then
        return true
    end
    // guess leads to some contradiction
    Cancel guess for next_cell // also restore it from neighboors that had it
end
// We have tried all candidates for this cell, without success
return false
```

Algorithm 1: Backtracking algorithm

As we can see the algorithm relies on a few underlying functions, that are expressed as methods (member functions) of the `Cell` class:

- A `Cell` should be able to tell if a given guess could be set as its digit
- A `Cell` should be able to set a guess and prune it from its neighboors
- A `Cell` should be able to cancel a guess and restore it for its neighboors

Additionally, a grid should be able to tell if it is full, that is all its cells have a digit.

3 Available classes and methods to implement

The `main` function is quite trivial and only loads the desired Sudoku to solve. The algorithm itself is done through two classes: `Grid` and `Cell` .

3.1 `Grid` class

The `Grid` class has:

- An array of 81 `Cell` called `cells`
- A constructor that takes a starting grid and should initialize the `cells` accordingly
- A `solve()` method that enters the solving process and prints the outcome
- A `solveNextCell()` method that implements the backtracking algorithm

You have to implement:

- the constructor that initializes the starting grid but does not yet erase candidate values for neighboors of cells defined in the grid
- the `solveNextCell()` method (shown in Algorithm 1)

3.2 `Cell` class

As seen above, most of the intelligence comes from the cell being able to change its guess and inform its neighboors. The `Cell` class has:

- A `init` function that takes the row and column of the cell, the whole `cells` array from the grid and any initial value this cell should have
- A `uint` called `digit_` that is the current guess for this cell. If `digit_` is 0 it means this cell was not assigned a value at this point of the algorithm
- A vector of `uint` called `candidates_` that lists the remaining candidates at any point during the algorithm
- An array of `uint` called `pruned_count_` that tells how many times a given digit was pruned for this cell. Index 0 is not used so for example `pruned_count_[3]` tells how many times 3 was said to be a forbidden value for this cell
- An array of 20 pointers to `Cell` called `neighboors_` that point to all other cells that cannot have the same digit as this one
- A member functions to erase a candidate during initialization: `eraseCandidate(guess)`
- A static functions `isValid(cell)` to be used in algorithms
- 2 member functions that are called by the backtracking algorithm: `set(guess)` and `cancel()`

- 2 member functions that keeps track of guesses in connected cells: `prune(guess)` and `restore(guess)`

You have to implement these four last methods, their definition being listed below

`void prune(uint guess)`

This method should delete the given guess from the candidates of this cell, if these two conditions are met:

- The cell does not have a digit yet
- The guess to be pruned is actually in the candidates, or the guess was already pruned

If the guess is actually in the candidates, it should be removed. In any case the corresponding value in the `pruned` array should be incremented to keep track how many times this was pruned.

`void restore(uint guess)`

This method should decrement the pruning counter of the given guess for this cell:

- The cell does not have a digit yet
- The guess was indeed pruned

If both these conditions are true, the counter in `pruned_count_` should be decremented. If this counter falls back to 0, the guess should be added again to the `candidates`.

`void set(uint guess)`

This method should set the guess on this cell and prune it from its neighboors

`void cancel()`

This method should cancel the current guess on this cell (available in `digit_`) and restore it from its neighboors

3.3 Tips

All the methods can be written in a few lines by using modern C++ that is:

- range-based for loops
- `<algorithm>` functions such as `std::any_of` and `std::all_of`, possibly with lambda or existing functions

Some basic errors will lead to the program crashing, **use the debugger** to understand what went wrong.
You can also use it with breakpoints to see what is going on in this mess.

Do not modify parts of the code you are not supposed to implement or you will never be able to solve a Sudoku again.

3.4 Bonus: improve the `bestNextCell` function

In `grid.cpp` the `bestNextCell` function takes two `Cell` called `c1, c2` and returns whether `c1` should be investigated before `c2`. The initial function just compares the cells' digit, ensuring that a cell with digit 0 (e.g. no guess yet) will always be chosen over a cell that is already set. This may lead the algorithm to start with cells that have a large number of candidates, while some others may have less and even only one. Improve this function as you think is best and compare the solving times.

3.5 Bonus: keep track of the algorithm back and forth

The `Grid` class has member variables called `guesses` and `cancels` that are printed at the end. Update them in the `solveNextCell` function to see what happens under the hood.

4 Starting grids

The `starts` folder lists several starting grids, to be set in the `main` function to test your algorithm:

- `basic0` is already filled. You should start with that to test the exit criterion.
- `basic1` and `basic2` have respectively only 1 and 2 empty cells
- `easy`, `medium`, `hard` and `harder` are of increasing complexity