Applied Programming (ID1218), 2009
Christian Schulte

Lab 2-2: Sudoku Solver

Intro date: 2010-11-30
Due date: 2010-12-07

Problem Description

You will develop a solver for Sudoku puzzles using propagation and search.

Lab Goal. The lab will familiarize you with the following issues:

- How to solve Sudoku puzzles (not the main goal, of course).
- How to devise explicit memory management schemes.
- How to use data structures with pointers.
- How to make use of some of the fine points of objects in C++ (such as default and copy constructors, destructors, assignment operators).

1 Sudoku Puzzles

A Sudoku puzzle consists of a 9×9 array of fields. Each field must contain one of the digits $1, 2, \ldots, 9$. In a *solution* some *constraints* on the digits in the fields must be obeyed: each row must contain all the different digits; each column must contain all the different digits; each major 3×3 block must contain all the different digits. An instance of a Sudoku puzzle is a square with some of the fields pre-assigned. Solving the puzzle is to find digits for all the fields such that all constraints are satisfied and that the pre-assigned fields are respected. A valid Sudoku puzzle has exactly one solution. In Figure 1 a Sudoku instance is shown. The major blocks are marked by having double-lines between them.

2 Solving Sudoku Puzzles

Solving Sudoku puzzles uses a 9×9 *board* of *fields*. A field is a set of digits between 1 and 9 describing what are the still possible digits for the field. A solution is computed by interleaving *propagation* and *search*.

			2		5			
	9					7	3	
		2			9		6	
2						4		9
				7				
6		9						1
	8		4			1		
	6	3					8	
			6		8			

Figure 1: A Sudoku puzzle

Propagation. Assume that a field on a board is a singleton set $\{d\}$ (the field is *assigned* to the value d). It is clear that none of the fields in the same row, column, and block can be assigned the digit d. This information can be propagated as follows: prune the value d from all fields on the same row, column, and block. When a field becomes empty by pruning, the Sudoku puzzle has no solution (this will be important when using search to find solutions). When a not yet assigned field becomes assigned to the value d' by pruning, then propagation is repeated for this field with the value d', and so on.

Solutions. If all assignments are propagated as described above, the constraints for Sudoku are always satisfied. The first step in finding a solution to a Sudoku puzzle is to assign all fields (and propagate the assignments) as given by the puzzle. For example, this requires to assign all fields where digits are given in Figure 1.

A board where all fields are assigned is then a solution to a puzzle.

Search. Assigning, pruning, and propagation might not be sufficient to find a solution as it can result in a board with unassigned fields. In order to find a solution recursive search is used as follows. Suppose that there is a not yet assigned field with digits $\{d_1, \ldots, d_n\}$ $(n > 1)$. Then we first try to assign the field to d_1, perform propagation, and then continue with search recursively.

What can happen is that propagation results in failure (a field becomes empty). In that case search fails and needs to try different digits for one of the fields. As search is recursive, a next digit for the field considered last is tried. If none of the digits lead to a solution, search fails again. This in turn will lead to trying a different digit from the previous field, and so one.

3 Implementing Fields

The first step is to implement fields. The file `field.h` contains a class definition `Field` where you have to complete the code for the member functions. The digits in a field are implemented as a bitset stored in an `int`: the digit `d` is included in the set if the bit `1<<d` is set. In addition, a field also maintains how many digits it contains.

You also have to complete printing of fields contained in `field.cc`.

Reminder: Member functions defined in a class definition are inline functions!

4 Implementing Propagation

Both propagation and search are implemented as member functions of the class `Board` (contained in the files `board.h` and `board.cc`). To implement propagation proceed as follows:

1. Implement constructors (in particular the copy constructor), assignment operator, and printing.

2. Implement the assign member function: it first checks whether d can be still assigned to a field at position $\langle x, y\rangle$ on the board (that is, d must be still included in the field). If not, false is returned. If yes, it checks whether the field is already assigned (in this case, it already has the value d). If the field is already assigned to d, true is returned. Otherwise, the assignment is made and propagated by calling the propagate member function and returning its result.

3. Implement the prune member function. If a field becomes assigned after pruning, this also needs to be propagated by the propagate member function. Pay attention to the fact that a value to be pruned can already be excluded from a field (in this case, no further propagation is needed and allowed. Why?).

4. Implement the propagate member function: just perform prune operations on all fields in the same row, column, and block.

You will find two different versions of the above mentioned member functions. For the time being, only implement those not taking an argument of type `Trail&`.

5 Implementing Search: Copying

You will implement two versions of search: one using copying and the other using trailing. As copying is simpler, you should try this first.

Complete the member function search (again, the one without a `Trail&` argument). It returns a pointer to a `Board`. If the result is `NULL`, no solution has been found. Otherwise, the pointer points to a solution board.

Copying works as follows: before a digit d is tried by assigning a field to it, you have to create a new board being a copy of the current board (use the copy constructor and allocate the board from the heap with `new`). Only then perform the assignment on the copy. If the recursive call to search returns unsuccessfully, a new board can be created for the next digit to be tried.

A particularly tricky aspect is to delete the boards created during search. A board created as copy must be deleted after recursive search starting from that board has returned failure. A board created as copy must not be deleted if recursive search returns this board as solution. You might want to postpone deletion of boards until everything else works.

6 Implementing Search: Trailing

The idea of copying is to always keep an original and perform assignments on a copy. Trailing works on a single board: a *trail* records changes to fields and allows to undo these changes.

A trail contains entries and marks. An entry stores a pointer to a field and a field. Trailing a field creates a new entry on the trail that stores the address of the field and its content. Undoing an entry restores the value of the field by writing back the field to the location as defined by the pointer.

Before search assigns a field, a mark is put on the trail. Each time a field is assigned or pruned it is trailed before it is changed. When search results in failure all trail entries up to the last mark are undone and the mark is removed.

The file `trail.h` contains the class definitions `TrailEntry` and `Trail`. Complete them. The trail is implemented as a linked list of trail entries where the trail remembers the last entry (that is, entries are maintained in stack fashion). Marks are implemented as entries where the pointer to the field is `NULL`.

Now complete all member functions of the `Board` class in `board.h` using a trail.

7 Running the Solver

A main program with many different Sudoku puzzles is already available. After running `make`, execute `sudoku.exe -help` for help on how to run the solver.

You can test that your solution has no memory leaks by giving a repeat argument on the commandline. Repeat the solving very often and check

how much memory is consumed (use `top` or the Windows Task Manager to find out).

The file `2-2-solutions.txt` contains the printed solutions for all example Sudoku puzzles. Moreover, the file lists how often search has been invoked (that is, if the number of invocations is 1, only propagation has been used).

8 Extensions

If you want to improve your Sudoku solver here are some ideas you can try (they are not mandatory but just for your fun):

Stronger propagation Propagation is only performed if a field becomes assigned. Assume that a field f_1 in a row stores the digits $\{d_1, d_2\}$ and another field f_2 in the same row stores the same digits $\{d_1, d_2\}$ (the same holds true for columns and blocks). Then no other field in the same row can take the digits d_1 and d_2: these digits can be pruned from all other fields.

This idea is based on the notion of a *Hall* set: a Hall set H is a set of fields $H = \{f_1, \ldots, f_n\}$ such that all fields have the same set of digits $D = \{d_1, \ldots, d_n\}$ (that is, it contains n digits). The set of digits D can be pruned from all fields not included in the Hall set H. There exist efficient methods to do propagation based on Hall sets, however they are beyond the scope of this assignment (see below). Maybe you want to try using Hall sets of size two and three.

Search heuristics Search picks an arbitrary field. A good heuristic likely to decrease the amount of search is known as *first-fail*: first pick a not yet assigned field that has the fewest number of digits left. Why is this good?

More efficient field selection How can you speed up finding not-yet assigned fields for search?

Last digit optimization When assigning a field during search to the last possible digit, it is not necessary to create a copy (or to put a mark on the trail). If the assignment does not lead to a solution, the original is not longer needed (or does not require restoration) as search will continue with the field tried previously anyway. How would you deallocate boards with copying and last digit optimization?

Timestamps for trailing When using trailing, the same field can be trailed multiply when performing propagation for the very same assignment performed by search. This is possible as more than just one digit can be pruned from the same field. However, trailing a field just once is

sufficient: only the original state of the field must be restored. Intermediate states during propagation are not interesting.

A technique to guarantee that a field gets trailed at most once is based on *timestamps*: each field and also the trail gets a timestamp (an integer). Whenever a mark is put on the trail, its timestamp is incremented. When a mark is removed, its timestamp is decremented. When a field is to be trailed, it is checked whether the field has a timestamp that is smaller than the trail's timestamp. If yes, the field is trailed and its timestamp is set to the trail's timestamp. Otherwise, the field does not require trailing: it has already been trailed since the last mark had been put on the trail.

Background: Constraint Programming

Constraint programming is a general method for solving arbitrary problems by propagation and search. Constraints are not restricted to constrain fields (or variables as they are called in constraint programming) to take different values. Arbitrary constraints useful for many problems are possible. Constraint programming is a very succesful technique for solving hard problems in areas such as scheduling, personal allocation, time tabling, and so on. If you are interested to use constraints for more than just solving Sudoku puzzles, check the course Constraint Programming (ID2204):

`http://www.ict.kth.se/courses/ID2204/`