Term Project: Sudoku Helper

CSCI 4526 / 6626

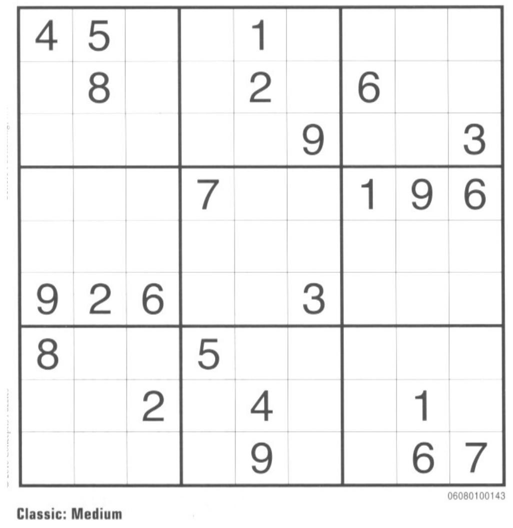
1 Project Overview

Solving Sudoku I like to solve Sudoku puzzles. Usually, I can solve an easy one without writing down anything but the answer. However, I can't solve hard puzzles that way; I need to make notes about my deductions to help me make more deductions. I can't keep it all in my head. So I have developed a variety of strategies for writing little numbers in the Sudoku boxes to tell me what might or must be in each box. My strategies work pretty well except for three problems:

- It gets pretty messy writing down and erasing all those little numbers.
- I make mistakes! and each mistake throws off all subsequent deductions.
- Puzzle-makers have now developed variations on the basic Sudoku that are even messier and more difficult than the original type of puzzle. See: http://www.conceptispuzzles.com/index.aspx?uri=puzzle/sudoku

The Project This project will not *create* new puzzles – many sources of puzzles are "out there". It will not solve a puzzle for you – that would be no fun. The goal is to allow you to input a puzzle easily, then interact with it until it is solved. The computer will do all the routine bookkeeping for you and check whether your moves make sense at the most basic level. To do this, we will implement a complex data structure and use as many parts of the C++ language as I can work into the plan. For the first part of the term, we will focus on and implement the basic, Classic style of puzzle, as shown above, on the left. After that, we will add variations. Students wishing to complete a "project out of a course" in this course may implement an additional, more complex variation, add the capability of generating random puzzles, or add functionality to the GUI interface.

Figure 1. A Classic Sudoku Board



Weekly Work I have designed a solution to this problem that involves several programmer-defined classes that work tightly together with each other and with library classes. We will use my design to talk about objects, classes, encapsulation, code-safety, and many other language features and design issues. Each week, I will ask you to implement part of this design – to create a new class or add functions and/or data members to an existing class. As soon as you have a stable and debugged *model* of the Sudoku game-board, I will supply a GUI interface to work with it. Since each of us will be writing part of the code, it is essential that you follow the instructions exactly and use the function names I specify. When you are finished, you will have a working interactive puzzle-game.

Classic Sudoku puzzles. The Classic Sudoku puzzle is a grid of squares with 9 rows and 9 columns, also divided into 9 square boxes. See Figure 1. Some of the squares have numbers written in them initially, but most are blank. (More numbers initially makes the puzzle easier to solve.) The squares of the completed puzzle are filled with the digits 1. . . 9 in such a way that every digit appears exactly once in every row, column, and box. To solve a Sudoku puzzle, you use two facts:

- Every row, column, and box must have one of each digit.
- Every time a digit is written in a square, it eliminates the possibility that the same digit will be written anywhere else in the same row, column, or box.

There is a unique solution for each puzzle, that can supposedly be found without guessing. In an easy puzzle, you can use single-step reasoning to steadily fill in one square after another. In a very-difficult puzzle, progress is harder to make and often three constraints must be used together to determine the contents of a square.

Sudoku Variations. Some Sudoku variations add more constraints to the puzzle, allowing the puzzle-maker to erase more of the initial clues without making the puzzle easier. Each new kind of constraint allows the solver to invent new solution strategies. One kind of constraint is to add diagonals. In a diagonal Sudoku, each diagonal must contain each digit exactly once.

There is also a new version of Sudoku that uses a 6x6 board that has columns, rows, vertical boxes, and horizontal boxes. These puzzles are substantially easier to solve. I will probably include them in the assignment as a derived class.

2 The Vision Statement.

We want to model a Sudoku game. The playing board has N rows with N squares in each row. Initially, each square either has a number (stored there when the puzzle was created) or it is blank. A Sudoku player attempts to fill the empty squares with digits $1 - -N$ so that no row or column has two squares with the same digit. In addition, boxes are superimposed on the playing board. In traditional Sudoko, each box has 3 rows and 3 columns. In sixy-Sudoko, each box is 2 rows with 3 columns or 3 rows with 2 columns. When the solution is finished, each box must contain N different digits.

At any stage in the solution, there are between 0 and N possible digits that could be written in a particular square. Writing a digit in one square removes that digit from the list of possibilities for all other squares in the same row, column, or box. We would like our implementation to provide an efficient way to determine whether a player's move is legal or not, and to provide a meaningful error comment if the player tries to make an illegal move. We want the game to tell us when we have won it.

When solving a Sudoku puzzle by hand, people do often see that a digit is wrong and erase it. But if that is done, all neighboring squares must be restored to their state before the error was made. We would like the application to have "undo" capabilities, which requires us to store a series of board-states. Of course, digits must be erased in the reverse order that they were entered, and the digits that were part of the original puzzle cannot be erased.

Sometimes these puzzles are quite difficult, and even a skilled player does not know what to do next. So he guesses. But before guessing and spoiling the board, he makes a copy of it (photo, Xerox). We want to be able to make a copy of the state of the game and return to that state later.

Finally, there are some variations on Sudoku that I would like to cover, after the basic game is finished.

3 Analysis.

1a. Find the Objects. The first part of an analysis is to identify and name the classes and the functionality that will be needed. Classes are used to represent objects, sets of objects, and processes. Looking at the vision statement, I see these nouns (classes or primitive types):

- The game or game variant: including shape of the puzzle and rules for solving the puzzle.
- Puzzle or board: the playing surface.
- Square: one component of the board.
- The value of a square: a digit $1 \ldots N$.
- Rows, columns, and boxes (groups of nine squares).
- Possibilities: a set of 1 or more digits between 1 and N that are still legal to put into a given square.
- Files for save and restore: strings for the file names, streams for the I/O.

Most of these things cannot be represented by primitive values, for example, a puzzle or board is much more than a single integer. These will be modeled by classes; the first step is to name the classes. In the process of developing the program, we may discover the need for additional classes.

The possibilities could be represented several ways: as a string of digits, an array of small integers, or a bit vector. The string of digits would be the bulkiest, a small integer would be the most efficient for space.

If everyone uses the same names for these classes, communication becomes easier and my job is easier. So please do.

2. Find the Functions. I see these action verbs (functions) in the vision statement:

- Input a puzzle or board.
- Fill an empty Square.
- Remove a value from the list of possibilities for neighboring squares.
- Efficiently determine whether a move is legal.
- Provide meaningful error comments.
- Undo a move . . . in reverse order
- Recognize a value that cannot be erased.
- Save the state of the game to a file.
- Restore the game state from the file.
- File names for save and restore.
- Recognize and announce a complete puzzle.

4 Use Cases for the Project.

These specify how the completed project should work. Modules will be built one at a time. Read the use cases now and refer to them later as you develop the relevant modules.

1. Load Traditional Sudoku Puzzle ————————————————————————-

Actors and their interests:

- Human player: Needs to load a puzzle from a file into the application, then solve it.
- Sudoku Helper, the application: Must validate that the file contains a legal sudoku puzzle. Needs to read the input file and store the file type (one letter) and initialize the contents (a character, digit or blank) of each square in the puzzle.

Preconditions:

- All of the information about a puzzle is stored in a file, including the puzzle type the initial contents of squares.
- The contents of the squares will be stored in the input file as N lines of N keystrokes (a digit or a dash). Each line ends in a newline. The constant N is 9 for traditional Sudokus and 6 for the modern variation.

Main Success Scenario:

1. Sudoku Helper prompts the user for a file name.
2. Solver supplies the pathname of a file relative to the directory that stores the executable program.
3. Sudoku Helper opens the file and reads the puzzle's type-code on the first line. If it is one of the currently-supported types, continue.
4. Sudoku Helper reads N lines of the file as strings and stores the input values into the N^2 squares of the puzzle. Each string has N characters, either a '-' or a digit 1…9.
5. For each digit input, Sudoku Helper adjusts the possibility-lists of all neighboring squares (row, column, box) to eliminate the new digit from the neighbors' possibility lists.
6. Any file content after the Nth lines is ignored. Sudoku Helper closes the file.

Deviant Scenarios:

1. (Step 3a.) File does not open successfully: Abort and display an error comment containing the name of the file.
2. (Step 4a.) First line of file does not contain a valid puzzle type-code (letter 't' or 'd' and dimension, 9 or 6): Abort and display an error comment that shows the legal type-codes.
3. (Step 5a.) EOF occurs before reading the contents of N^2 squares: Abort and display error comment.

Use Case 2. Mark a Square

Actors and their interests:

- Human: Enter a digit into a square as part of the solution to the puzzle.
- Sudoku Helper: When human enters a valid digit into a selected square, helper must adjust the possibility lists of neighboring squares to maintain a consistent state for the whole puzzle. A record of this transaction is kept to allow UNDO operations.

Precondition: The puzzle has been successfully loaded into the application.

Main Success Scenario:

1. Solver selects the menu option "*Mark squares*" and enters a digit.
2. Sudoku Helper prompts for one or a series of squares and stores the given digit in each.
3. Sudoku Helper eliminates this digit from the possibility lists of all neighboring squares (in the same row, column, or box).
4. Sudoku Helper redisplays the changed game board.

Deviations:

1. (Step 2a.) The given digit is not in the possibility list of the selected square: A very visible error comment is given and no change in state happens. The game continues.
2. (Step 3a.) When the selected digit is eliminated from the neighborhood, some square is left with zero possibilities: A very visible error comment is displayed. The change that caused this problem is automatically undone. The game continues.

Use Case 3. Turn Off a Possibility

Actors and their interests:

- Human: Eliminate a digit from the possibility lists of a series of squares.
- Sudoku Helper: Turn off the given digit from all selected squares. If any action results in a change in the possibility list, keep a record of the action to allow UNDO operations.

Precondition: The puzzle has been successfully loaded into the application.

Main Success Scenario:

1. Solver selects the menu option "*Turn Off Possibility*" and enters a digit.
2. Sudoku Helper prompts for one or a series of squares and eliminates the given digit from the possibility list of each.
3. If the elimination changes the possibility list, the number of remaining possibilities for this square is decremented and a record of the change is made to support *Undo*.
4. Sudoku Helper redisplays the changed game board.

Deviations:

- (Step 2a.) When turning off the selected digit causes a square to be left with zero possibilities: A very visible error comment is displayed. The digit is NOT turned off in that square. The game continues.

Use Case 4. Save ————————————————————————

Save the partially-solved puzzle so that it can be finished later. Use a file name given by the Solver. Permit the Solver to continue working on the puzzle.

Use Case 5. Restore ————————————————————————-

Restore a previously-saved puzzle from a file so that it can be finished.

Use Case 6. Undo ————————————————————-

Revert to the puzzle state preceding the most recent move. The Solver can back up as far as the most recent Save operation.

Use Case 7. Redo ————————————————————-

Reverse the actions of an Undo operation.

Use Case 8. A Variant ————————————————————-

The last use case for the Sudoku variant will be announced in mid-term.