Term Project: Sudoku Helper

CSCI 4526 / 6626

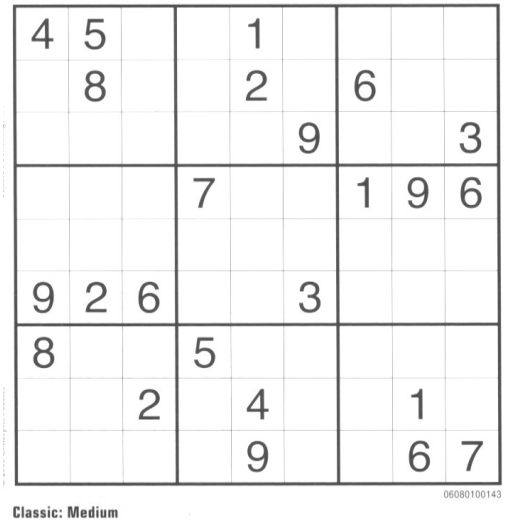
1 Project Overview

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 color a subset of the squares and specify that all of the colored squares must contain odd digits (or even digits). Figure 2 shows an Odd-Sudoku puzzle. Figure 3 is an Even-Sudoku.

Figure 2. An Odd-Sudoku Board

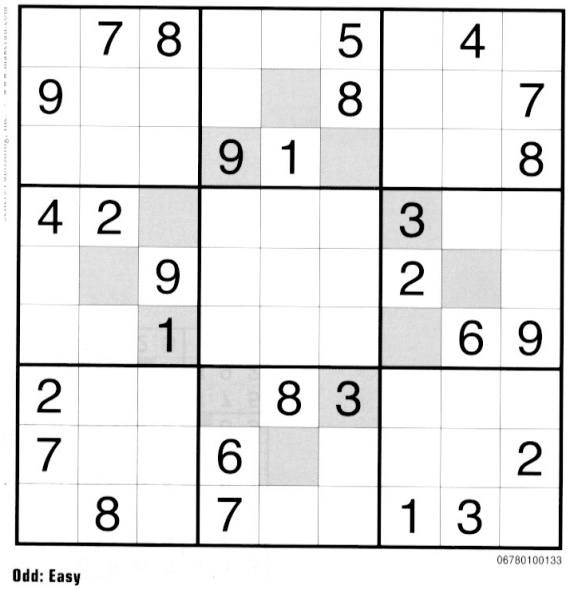
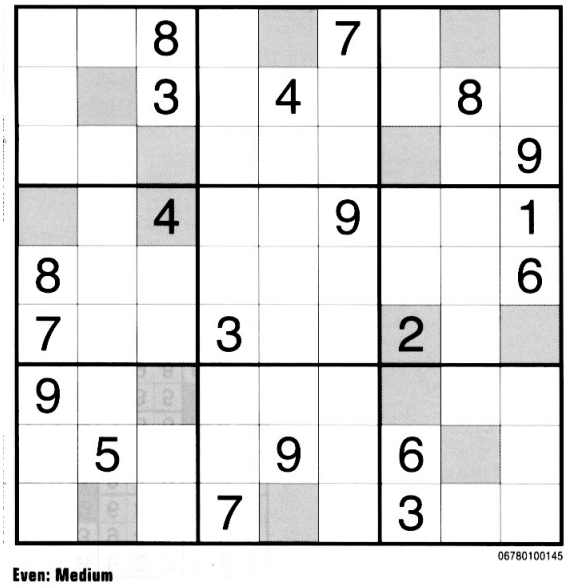


Figure 3. An Even-Sudoku Board



Another constraint is to add diagonals, as in Figure 4. Each digits must occur once on each diagonal. A diagonal constraint could also be combined with an odd or even constraint, although we will not consider the combinations as part of this project.

Figure 5 shows an Irregular-Sudoku puzzle. Instead of rows, columns, and 3-by-3 boxes, it has rows, columns, and 9-square snakes. This does not add constraints, but it mixes them up, forcing the solver to invent flexible solution strategies. The goal for the term is to implement all five of these variations.

Figure 4. A Diagonal-Sudoku Board

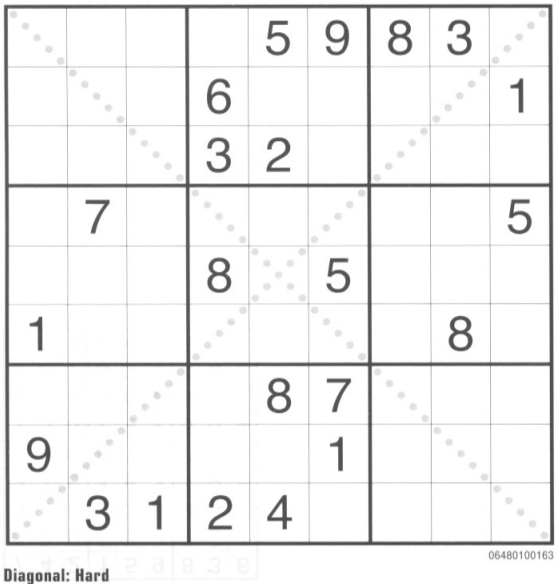
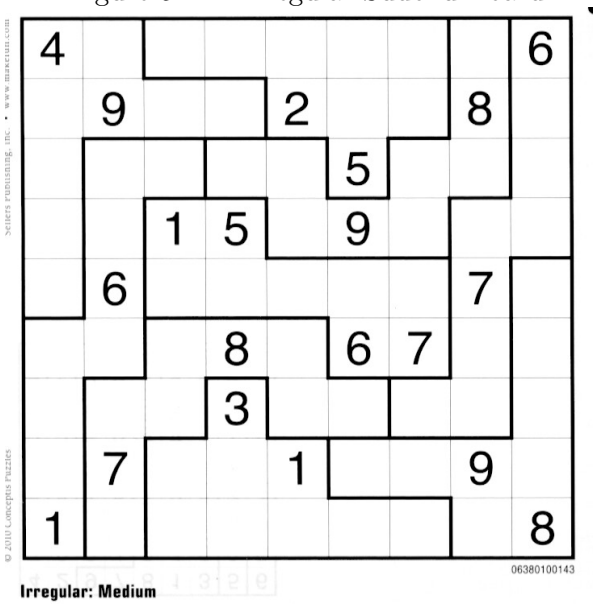


Figure 5. An Irregular-Sudoku Board



Advanced Variations. The website for Conceptis Puzzles lists many additional variations, including the following two that are recommended for students wishing to do a project-out-of-a-course.

Figure 6 shows a Sum-Sudoku Puzzle. In addition to the row, column, and box constraints, the board is subdivided into smaller irregularly-shaped sets of squares, outlined by dashed lines. The digits in each set must sum to the value written in the upper-left corner of the area. No digit may be repeated in any sum-area. Even an easy sum-sudoku can be difficult. Solving such puzzles would be much easier with the help of a computer that could generate lists of all digits that are still possible in an area and that sum to the required number.

Figure 6. A Sum-Sudoku Board

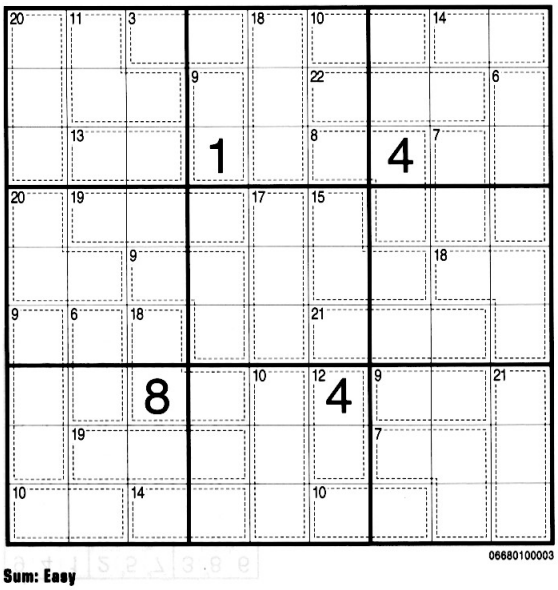


Figure 7. A Multi-Sudoku Board

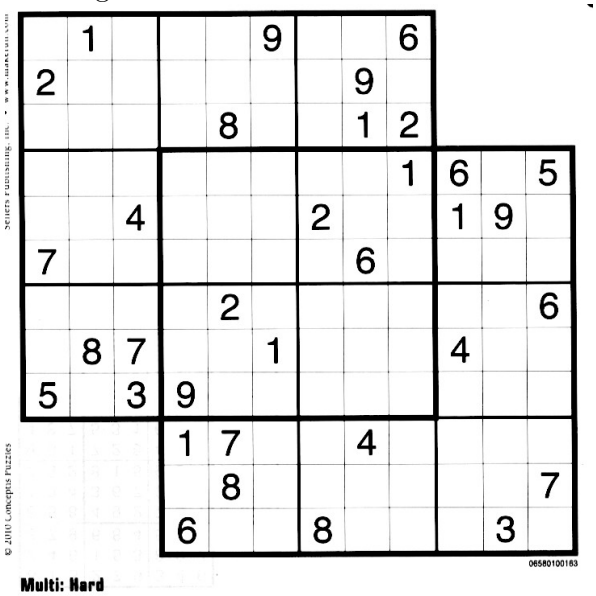


Figure 7 is a Multi-Sudoku puzzle: two squares superimposed. In these puzzles, all the rules for rows, columns, and boxes still apply, but the central four boxes are part of both puzzles. The Conceptis website also shows triple-sudoku puzzles and other complex overlapping arrangements.

2 Use Cases for Project Phases 2 and 3

1. Load Traditional Sudoku Puzzle

- Actors and their interests:
  - Puzzle solver, a human.
    Solver needs to load a puzzle from a file into the application.
  - Sudoku Helper, the application:
    Must validate that the file contains a legal kind of sudoku puzzle.
    Needs to read and store the initial contents (a digit or blank) of each puzzle square.
- Preconditions:
  - All of the information about a puzzle is stored in a file, including the puzzle type, the initial contents of squares, and possible additional constraint information.
  - The contents of the squares will be stored as nine lines of nine keystrokes, ending in a newline.
- 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 9 lines of 9 characters and stores them into the 81 squares of the puzzle. Input is read in such a way that whitespace is NOT ignored, except for the newline character that ends each line.
  5. For each non-blank input value, Sudoku Helper adjusts the possibility-lists of all neighboring squares (row, column, box) to eliminate it from the neighbors' possibility lists.
  6. Any file content after the nine lines is ignored. Sudoku Helper closes the file.
- Deviations:
  - 2a. File does not open successfully: Abort and display an error comment containing the name of the file.
  - 3a. First line of file does not contain a valid puzzle type-code: Abort and display an error comment that shows the legal type-codes.
  - 4a. EOF occurs before reading the contents of 81 squares: Abort and display error comment.

2. Mark a Square

- Actors and their interests:
  - Solver: Enter a digit into a square as part of the solution to the puzzle.
  - Sudoku Helper: Enter a valid digit into the selected square and adjust the possibility lists of neighboring squares to maintain a consistent state for the whole puzzle. Keep a record of this transaction 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:
  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.
  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.

3. Turn Off a Possibility

- Actors and their interests:
  - Solver: 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 action is made to support UNDO.
  4. Sudoku Helper redisplays the changed game board.
- Deviations:
  2a. 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.

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.

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

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.

7. Load Odd or Even Sudoku Puzzle Allow the entries in the puzzle-file to be digits or the character 'G', for gray. Initialize non-gray squares as for traditional puzzles. Initialize gray squares by turning off all the odd digits or all the even digits in the possibility list.

8. Load Diagonal Sudoku Puzzle To be specified later.