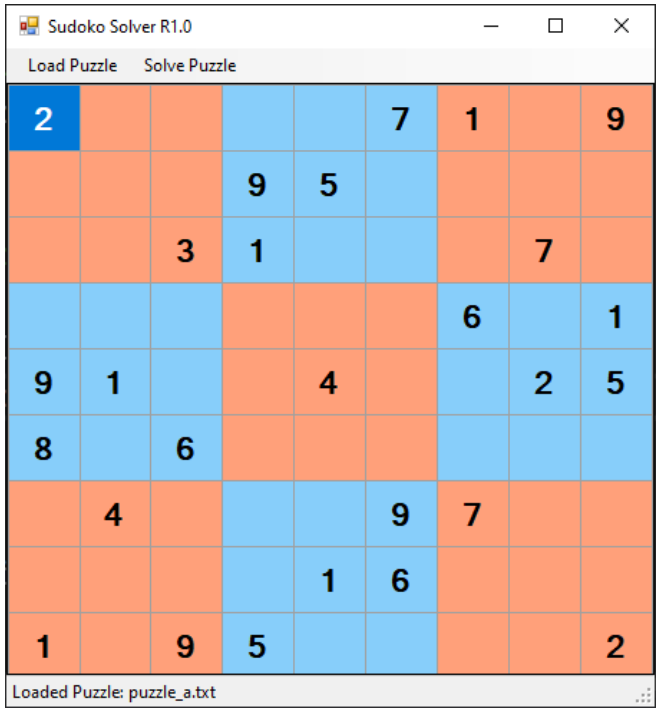
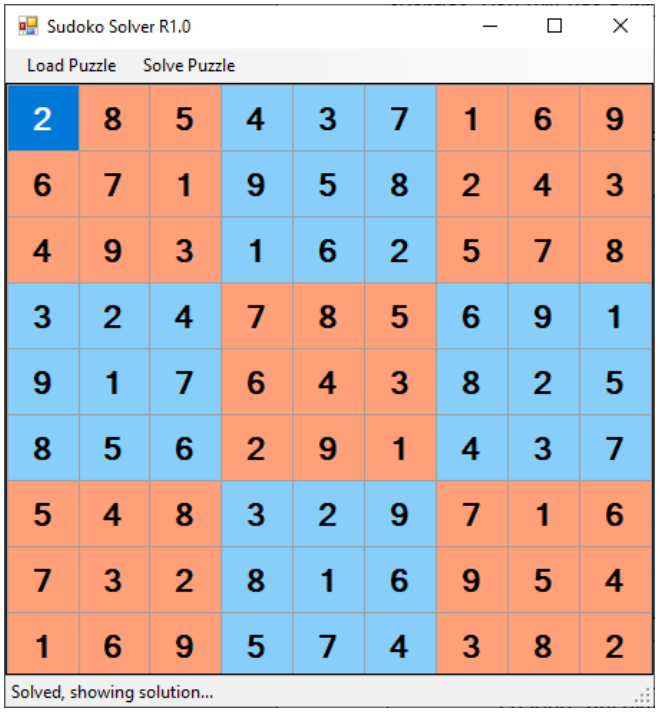
CMPE2300 – Lab #01 – Review – Sudoku-matic

In this lab you will create a Sudoku solver. While many approaches are possible, as a review exercise, you will use a brute-force recursive strategy. The puzzles to solve will be loaded from a text file, and sample puzzle files will be provided. The program will load the puzzle from the file and display it to the user. The user will have the option to solve the puzzle and display the solved puzzle.



Loaded, unsolved puzzle.



Solved puzzle.

Program Specification

The application will appear as shown above. The UI will consist of a MenuStrip at the top, StatusStrip at the bottom, and a DataGridView as the bulk of the client area to display the puzzle.

The MenuStrip will contain two options:

- Load Puzzle – load a puzzle from file into the application
- Solve Puzzle – if a puzzle is loaded, attempt to solve it

The StatusStrip will contain a single label that will show status information to the user.

The DataGridView will be used to display the loaded and solved puzzles.

In your form constructor, bind all UI elements to the desired event handlers – do not bind them in the event property inspector.

You will need a data member to hold the puzzle information. You will use a 2D rectangular array of integer. This will be the only data member you will require. Use a value of zero to indicate an empty cell, or an integer from 1 – 9 to represent a filled cell.

The StatusStrip will be used to feedback useful information to the user. This will include operational messages as well as error messages. You will additionally direct detailed debugging messages, as appropriate, to the System.Diagnostics.Trace stream.

Loading Puzzles

Several puzzle files with various configurations will be made available to you. Some puzzles will normal, will have errors in the format, or may not be solvable. Expect additional files to be made available at check-off time.

The general format for the puzzle text file will be:

```
[[5,0,3,0,0,7,0,0,0],
[0,0,1,0,0,0,0,8,0],
[8,4,0,3,0,9,0,1,0],
[0,0,0,2,0,0,0,0,7],
[0,7,0,5,0,4,0,6,0],
[1,0,0,0,0,6,0,0,0],
[0,2,0,9,0,1,0,5,4],
[0,1,0,0,0,0,6,0,0],
[0,0,0,7,0,0,2,0,1]]
```

Your instructor will discuss strategies for parsing and loading the puzzle files.

The Load Puzzle menu item event handler will do the following:

- Use a locally created OpenFileDialog to have the user select the desired puzzle file. To make it easy to find the sample puzzles, set the initial directory for the dialog to the executable location (Directory.GetCurrentDirectory()). Set the dialog filters to include puzzle files (*.txt), and all files (*.*).
- If the dialog is closed with result OK, call a helper method called 'LoadPuzzle', otherwise do nothing.
- If LoadPuzzle is successful, update the status to reflect this. Ensure that you show the puzzle filename, as “Loaded Puzzle: nnn.txt”
- Call another helper method called 'ShowPuzzle' to display the loaded puzzle.

As discussed above, you will require some private helper methods:

LoadPuzzle() - returns a bool to indicate the outcome of loading the puzzle. Accepts a string as the filename to load.

Use System.IO.File>ReadAllText() to load the full contents of the selected file into a string. Don't forget error checking with file operations!

You will need to parse the string to extract the data for the 2D grid that makes up the puzzle. Basic error checking will include checking for framing characters, numbers in range, missing components, etc...

If all works well, you will have a 9x9 rectangular array loaded with integers.

Your instructor will discuss strategies for parsing and loading the contents of the puzzle file.

ShowPuzzle() - Displays the 9x9 rectangular array to the DataGridView. There are many ways to add data to the DataGridView to provide the appearance that you are looking for. The following is a simple way to achieve what is shown in the sample above.

- Set the column count to 9. You can do this only once, so doing it in the load event would be OK.
- Set the AutoSizeMode for each Column to Fill. Again, you can do this once.
- Clear() the Rows of the DataGridView.
- Create an array of 9 strings that will represent a full row of data. Populate with the numbers for that row.
- Add() the array of strings to Rows of the DataGridView.
- Adjust the properties of the Rows you just added (Alignment, BackColor, ForeColor, Padding, Font).
- Repeat for the other 8 rows.
- Alternatively you may adjust the properties of Rows[y].Cells[x].Style after all the rows are added.

There are properties of the DataGridView that you might want to change at design time that will contribute to the appearance you want.

NOTE: You do **NOT** need to have the DataGridView fully and dynamically resize to the window. You might be able to get some aspects of this working, but it is not a requirement.

You may want to create a separate project as a sandbox to experiment with DataGridView behavior and to conduct testing.

Fully test the load and show before you continue. You might also find it useful to have a debugging function that can dump the puzzle to the debugging console. This will be useful when you are writing the solve code, so that you can see intermediate steps in the solve.

Solving the Puzzle

Solving the puzzle will be done recursively. You will attempt to put every legal value into every empty (0) cell. You will add an event handler to the Solve Puzzle menu item to get things started.

Manage your UI to only enable the Solve Puzzle menu item if a puzzle is loaded.

You will need another private helper method called ValidSpot. It will accept a row, column, and a value to test. The function will return true if the test value is unique in the row, the column, and 3x3 sub block of the grid, otherwise it will return false. The recursive function will use this helper to determine possible acceptable values for a solution.

The recursive SolvePuzzle function will attempt to solve the puzzle. It will return a bool indicating if the puzzle was solved or not. Using nested loops, you will visit all 81 cells of the puzzle. For cells that are empty (0), you will test each possible number from 1 to 9 with the ValidSpot helper function. If the spot is valid, you will put that number in the current cell, and recursively call the SolvePuzzle function. If the SolvePuzzle call returns true, return true from the function, otherwise set the cell back to empty. If, after testing each valid number in a cell, there is no valid spot, return false (this will occur naturally if the recursive calls never return true). If you visit all 81 cells and never returned false, then no cells are empty, and you win, so return true.

Your instructor will provide additional explanation of this recursive functionality, as it is quite different from the format you have seen in previous courses.

Rubric

Basic UI, Load (basic), Show (all to specification)	30
Recursive Solve	50
Load (all error conditions handled correctly)	20

Marks will be reduced for failing to follow standard documentation and coding standards. Proper documentation is critical in your final solution. Programmer's blocks are required on all .cs files.

Code as incrementally as possible, validating as you go. Any milestone, fix, or victory should be committed to SVN with an appropriate comment.

Any submission that does not have logged evidence of progress will **not** be accepted.