Sudoku Solver

by Matt Breckon

Process

Starting Fresh with Ada

Starting with a blank slate and no experience in Ada made this task an interesting one. I managed the entire process with git to track changes to the project; however, unlike the first assignment re-engineering a tic-tac-toe program, the commits were sparse covering mostly major additions to the sudoku solver. The following document will cover design decisions, the algorithmic process, and overall analysis of the development process.

At First Glance

To run the program, a file must first be input through the command line:

```
./sudoku [file_name.txt]
```

During the main execution, the user will be prompt to select an output for the solved puzzle: “1” for standard out to the terminal and “2” for file. If the user selects output to file, the user will then be prompt for a file name. Note: The program automatically appends “.txt” format to the output file.

At First Glance

The project began with starting with the basics of Ada while adapting similarities to other known languages. Overall, Ada felt a lot like C without the memory management. I began the project with handling file input and output. The first commit, “Added file I/O for loading boards and printing boards,” proved to be rather straightforward. Through the use of Ada packages such as Ada.Command_Line, Ada.Text_IO, and Ada.Integer_Text_IO, file management was a breeze.

The main process starts with checking argument count. To execute the program, a single file containing a sudoku board must be provided at the command line. This process could have been improved with error checking for valid files and formatting. After the argument count check, the board is loaded into the program.

A single function and two procedures were created for file input, standard out, and file output. Function load_board takes a single string representing the file name and returns a 2D 9 by 9 array of type sudoku_board. Procedure print_sudoku takes a single argument of

type sudoku_board and prints the given board to standard out. Procedure output_sudoku takes two arguments: a string representing the file name for output and a sudoku_board. This procedure then writes an identical output to the procedure print_sudoku to the given file name with an appended ".txt" format.

Design Structure

The second major commit to the project, "Added functions to check if an integer is legal in a given cell," consisted of four core functions: in_row, in_column, in_square, and is_legal. The concept here was to pass an integer to is_legal along with two coordinates, x and y, and a specific board. Function is_legal is a boolean function that returns true or false based on whether or not a given value is legal in the provided cell x, y. To determine whether or not a value is legal in a provided cell, this function calls in_row, in_column, and in_square.

Functions in_row, in_column, and in_square are straightforward boolean functions that take a specified value and row and/or column. Based on the rules of sudoku, these functions return true or false if a given integer is found in a given row, a given column, or a 9 x 9 given subsquare. Combining these three functions in is_legal can determine if an integer can be placed in a given cell.

The third major commit, "Added solving function for immediate solvable cells. Added stack ADT for boards," involved added three core functions to the program, along with a stack to manage working board states.

Function solve_single_cell was implemented to take a cell of parameters x and y, and a given board. This function runs through integers 1 to 9 to attempt to solve a cell of x and y by calling function is_legal. If no value is legal or more than one value is legal, the function returns 0 to signal that no answer was found to the given cell.

To expand on the previous function, the function solve_cells was implemented to take a board. This function loops through the given board's 9 by 9 array looking for unsolved squares. If an unsolved square is found, it calls solve_single_cell to attempt to solve it. This entire process allows for a potential solving of the entire board assuming each cell is solvable.

Finally, the most important boolean function solve_sudoku to automate the above functions was implemented. This recursive function tracks a current working cell x and y, along with a current working board. This function begins with a loop executing solve_cells;

the board is repeatedly attempted to be solved until no changes are made to the board. From here, the function calls a new function is_blank_squares to look for the first blank cell on the board; this sets the current cell x and y values to work with. If there are no blank cells on the board, this function assumes the board is complete and returns true.

The next step of the solve_sudoku function involves pushing the current board onto a stack and executing a loop of variable i from 1 to 9. This loop attempts to place the variable i into the current working cell x and y. If a value is placed in the cell, the function then recursively calls solve_sudoku to further attempt to solve the board with this brute force attempt at placing variable i in the working cell. This recursion will branch down until the board is either solved or is unsolvable. If the board is solved, the function will return true recursively passing back the completed board. If the board is unsolvable, it will return false recursively returning to a previous board state. The stack will pop off the previous board and try the next value within the loop from 1 to 9.

After everything above occurs, if the process reaches the end of the function, it returns false assuming the board is impossible to solve. This impossible state occurs when an input board has invalid values in invalid locations. Note: See Problems and Issues below.

Discussion

Problems and Issus

During the design process, an intial problem was faced when dealing with retrieveing user input. Using the function get_line, the program asks the user for a file name to output the results to. However, an initial call of get_line resulted in it being bypassed and the string being stored as a blank “”. By providing the an idential call to get_line immediately after solved this problem. Having minimal experience with Ada, I could not solve this problem other than a hacky-looking double call to the get_line function. I have suspicions that a call to the function get a few lines before conflicts with the following get_line character stream; however, I cannot confirm or deny this.

The second major issue involved with the project, and a major complication with the stack, was understanding how Ada handles variables. The conflict of “pass by value” and “pass by reference” resulted in issues with pushing the current working board onto the stack. Popping a board afterwards did not seem to result in retrieving the board on the stack but instead all of the boards on the stack seemed to point to the same address in memory. This in turn meant any puzzles that required the stack to solve it, such as boards board3.txt and board4.txt within the boards directory, became unsolvable. I attempted to fix this solution with a new function copy_board to directly clone a board; however, this did

not seem to solve the problem meaning a core misunderstanding of variables in Ada is my biggest problem with the program functioning correctly.

Structures and Recursion

Creating and working with data structures within Ada was as straightforward as C. Creating types and structures was as straightforward as creating structs within C. Recursion within Ada was also identical to any other languages that support recursive calls within functions.

Language Quality

Ada proved to admittedly be one of my least favourite languages so far in my programming career. More so due to a lack of understanding how to handle variables, and also due to the tedious nature of the language, such as using end statements instead of enclosing blocks with braces, in comparison to more straightforward languages such as C. I particularly did not enjoy how variable declaration was handled as well. What I particularly enjoyed about Ada was how simple packages were and the vast amount of packages that were readily available with the simple addition of “with Ada.Package_Name”. I also enjoy languages that do not require manual memory management; however, I do understand the power behind memory management.

Comparison to C

I am biased in answering the question, “would it have been easier to write the program in a language such as C,” as I had previously built a **working** sudoku solver in C for a course in my previous year. I did not face the previously mentioned issues when developing the program in C. However, had I known Ada as well as I knew C when I first attempted such a project, I feel that Ada would have been the more accessible choice due to the power behind the language.

Improvements

Although my implementation was efficient in coding simplicity, it was not efficient in solving performance. A much better way of writing this sudoku solver would be through the use of heuristics. My current implementation is a brute force solver that solves all possible immediate solvable squares, and then checks cell-by-cell if a given integer from 1 to 9 works. Performance on difficult puzzles can result in massive amounts of board states being generated in order to solve a puzzle, assuming the stack is utilized correctly.

Language Complexity

Given my knowledge of programming, I found Ada rather straightforward, excluding passing variables. It felt a lot like C in syntax. I found Ada's handling of passing values to be much cleaner than how C uses pointers; however, as demonstrasted, I did not properly utilize the in, out, and in out parameters while passing my board arrays.

Structures

Throughout the development process, I was not able to identify specific structures I had used found within Ada and not within C. Every structure I used within my sudoku solver for Ada was also used in a similar fashion within my C sudoku solver. A stack data structure was created as an external package just like an external stack library was used for in C. However, not necessarily a structure, but I particularly enjoyed the ease of use when declaring ranges for a variable type. This made loops very easy to utilize such as a for loop from 1 to 9.