Timothy Oltjenbruns
CS4242 Artificial Intelligence
Mike Franklin
Project 0: Sudoku

Sudoku is a logic game with simple rules and multiple skill levels. Each new skill level requires a more expanded set of reasoning than the previous. Each of these rules reduces the number of possible values that could be placed in a particular position. To model my approach with the concept of a decreasing number of possible values, I designed a data structure accordingly.

Methods

The data structure is a deciding factor in the success of my solver, so I tackled it first. Call it rule 0 if you like. One 9 by 9 array stores integer values and another stores singly linked lists. The first represents a paper that the puzzle occupies and the second represents the options that a human stores either in their memory or on a separate paper. To ensure integrity, only values of 0-9 are allowed for the first array, and 1-9 for the second. The class comes with convenience functions to check for a value or count option values in rows, columns, and nonets. After designing the structure, I wrote equality, stream input/output operators, and a function to count how many values differ between two puzzles. The last function is particularly useful for checking supposedly solved puzzles against known solutions.

I spent the bulk of remaining time on two functions that leveraged the data structure to define rules for the algorithm to place values. The finalize function checks a single options list against other related lists and attempts to place a number on the puzzle using rules 4-10 described later. The solve function uses rules 1-3 which populate the initial options lists, then calls finalize on every puzzle position until a complete pass is run without yielding any further placed values.

To populate the initial options lists, I simply generate the maximum possible list at every position. A value is added to the list if it does not already exist in the row (rule 1), column (rule 2), or nonet (rule 3). Thankfully, these checks are built in to the data structure designed earlier. Generating these initial lists help reduce the number of useless repeated calculations.

Once the options lists are generated, rules 4-10 start acting upon the puzzle. If any options list contains only one value, that value must belong in it’s position, and it is placed on the puzzle (rule 4). Placing a value on the puzzle removed the value from all other options lists in the same row (rule 5), column (rule 6), and nonet (rule 7).

The looping of rules 4-7 typically solves easy puzzles completely. However, rules 8-10 are introduced to solve medium puzzles. I call these the “solo” rules, unfortunately, they do not shoot first or have any relation to Han. When a value in an options list is the only instance of that value in a row (rule 8), column (rule 9), or nonet (rule 10), it must also be placed on that position. This also triggers rules 5-7 again. Looping rules 4-10 usually solves medium puzzles with high accuracy.

Experiments

Lucky for me and the records, I used version control on this project. Some changes and experiments were also logged using version control. Rules 1-3 came first, and yielded 10 placed numbers on puzzles/easy/133. Adding rule 4 tested with 32 or 42 placed values, 21 or 29 of which were incorrect. I removed rule 3 and 4, fixed some bugs, and reintroduced them along with new rules 5-7. Then I placed all 55 numbers with 0 incorrect. I started recording performance with medium difficulty, only placing 3 on puzzles/medium/221. After adding rules 7-10, I place 33 with 1 incorrect on puzzles/medium/209, 11 with 0 incorrect on puzzles/medium/221, and 58 (complete) with 0 incorrect on puzzles/medium/222.

Analysis

This project had its bugs and setbacks, but they were manageable. The bugs that introduced instability in rules 3 and 4 were definitely caused by some bad modulus math and variable misnaming. The column and row variables were switched and caused plenty of incorrect placements. Rules 5-7 were straightforward after fixing math errors. These rules were well thought out on paper first. Rules 8-10 solve some medium puzzles (222 for example), complete with error on some (221), and partially complete on others (209). Either these rules are bogus and just happen to work, or they still have a bug or two.

Conclusion

Most of my design and rules were tried and tested on paper before even programming the data structure which helped fit the solution to the task. Even rules 8-10 were thought of beforehand, though until I got to medium puzzles I thought they might be useless and redundant. If I did this project again I would have followed through writing unit tests on the data structure before writing the rules. Overall, the project was smooth and enjoyable while I learned how to better translate human thought to code.

PROJECT REFERENCES:

Source of Google Test included (as instructed by installation manual), obviously I did not write this.

CodeCoverage.cmake automated buildscript is derived from https://github.com/bilke/cmake-modules/blob/master/CodeCoverage.cmake

BUILDING:

1. run Cmake on the root directory to generate makefiles for your environment
2. run make or equivalent to build source using the previously generated makefiles
3. run “run_sudoku help” for more info on how to use the program.

Contains an option to make new puzzle files, and an option to run the algorithm on existing puzzle files.