General Description

Project Group Members : Amanda Steidl, Kyle Stanski, & Lucas Crocket

Course : Analysis of Algorithms

Professor : D'Antonio

Project : Final - Soduko Algorithm

Due Date : December 28, 2016

#### File structure

Steidl\_Stanski\_Crocket\_FinalSubmission

|-> BoxSet.cpp //File containing the *“better” algorithm*

|-> BoxSet.h //File containing the *header to “better” algorithm* (BoxSet.cpp)

|-> BruteForce.cpp //File containing *the brute force algorithm*

|-> BruteForce.h //File containing *the brute force header*

|-> easypuzzle1.txt //Testing file 1

|-> easypuzzle2.txt //Testing file 2

|-> easypuzzle3.txt //Testing file 3

|-> Source.cpp // File containing the programs main currently set to test Timing for BoxSet

|-> Sudoku.cpp // File containing the backbone for setting up the board from the files

|-> Sudoku.h // Header containing additional member variables for Sudoku.cpp

Sudoku algorithm

## The Approach

Coming at the problem we decided to think of how we each solve normal Sudoku problems.

* Our techniques revolved around the games general rules:
  1. Each section had to contain the numbers 1 thru 9
  2. Each row had to contain the numbers 1 thru 9
  3. Each column had to contain the numbers 1 thru 9
* To make it a bit more detailed we compared our strategies:
  1. Finding a row and/or column with only one “space” available : must be whatever number it’s missing.
  2. Finding a single digit missing from a box : must be the missing number.
  3. Process of elimination.
  4. Comparing boxes to one another, and finding violations.

The finalization of steps concluded with simple pseudo-code of how a computer could possibly handle the problem and what data structures to use.

* Arrays would be the easiest structure to use since there’s instant access as well as we could use the indexes as positions in the puzzle.

### Responsibility breakdown

* Group :
  1. Pseudo-code and brainstorming
* Lucas :
  1. Coding the Brute Force algorithm
  2. Testing Code & Debugging
* Kyle :
  1. Coding the “Better” algorithm
  2. Testing Code & Debugging
* Amanda :
  1. Assisting coding the “better” algorithm, while debugging.
  2. Wrote simple test files
  3. Testing Code & Debugging
  4. Timing the algorithms on different puzzles to see time differences / change.
  5. Documenting the code
  6. Write-up

### comparison

Notes:

* Time is in MICROSECONDS
* Each puzzle was solved in sequence within the same program. Hence the increase in time after puzzle one.
  + Each algorithm was run in the same fashion.
* Used chrono::high\_resolution\_clock
* Highlighting corresponds to which one was better.

|  |  |  |  |
| --- | --- | --- | --- |
| Puzzle # | Brute Force Time |  | “Better” Algorithm Time |
| 1 | **45623** |  | **46479** |
| 2 | **109339** |  | **125840** |
| 3 | **137026** |  | **128981** |
| 1 | **42429** |  | **34957** |
| 2 | **125861** |  | **107843** |
| 3 | **129716** |  | **99577** |
| 1 | **38498** |  | **34229** |
| 2 | **118754** |  | **104187** |
| 3 | **129274** |  | **97664** |
| 1 | **41435** |  | **44423** |
| 2 | **121329** |  | **123514** |
| 3 | **138376** |  | **145580** |
| 1 | **49991** |  | **35404** |
| 2 | **137365** |  | **100191** |
| 3 | **124729** |  | **104260** |

The algorithm we wrote was faster compared to the brute force algorithm we wrote, however, since we were using more data structures to keep track of options and conditions of a single box it takes up more memory than the brute force algorithm.

Pseudocode : brute force

isValid

(1) value not already in that row

(2) value not already in that column

(3) value not already in that boxSet

for each box

for value in range [1 to 9]

apply value and move to next box

if this is last box, return true.

if next box says value was bad

try the next value

else value was good

tell the previous box that the problem has been solved

* Note : Changes were made to the algorithm when we began testing for errors.

idea behind : “better” Algorithm

# “Boxset” approach

**class BoxSet**

* A box set is a 3x3 of the possible numbers 1 – 9

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 2 | 3 |
|  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

* What does it need to do?
  + be able to input a value to a position
  + be able to say it has a value at a position
  + check possible values at a given position by communicating with other boxsets
  + tell nearby boxsets when it comes up with a decision (I must have # in my row #2, so you guys can’t) (I must have # at my row2col3 so you guys can’t interfere)
    - implemented by allowing boxSets to remove possibilities from the rows & columns of other boxsets
* \* If coming across a boxset who has a box without possibilities we can declare the puzzle unsolvable.
* How do we know we are done:
  + have a counter: when a boxset declares it can only have a # at (row, col) solves at (row, col) and increment the counter ( which will be global to the class ) when the counter is 81 we are done.

input & output

* From text files easypuzzle1.txt, easypuzzle2.txt, & easypuzzle3.txt respectively.

INPUT (from file) :

0 8 0 4 0 1 0 2 5

0 5 0 8 0 0 7 3 0

0 6 4 0 2 0 0 0 0

6 0 2 0 0 0 0 8 0

0 0 5 0 0 0 2 0 0

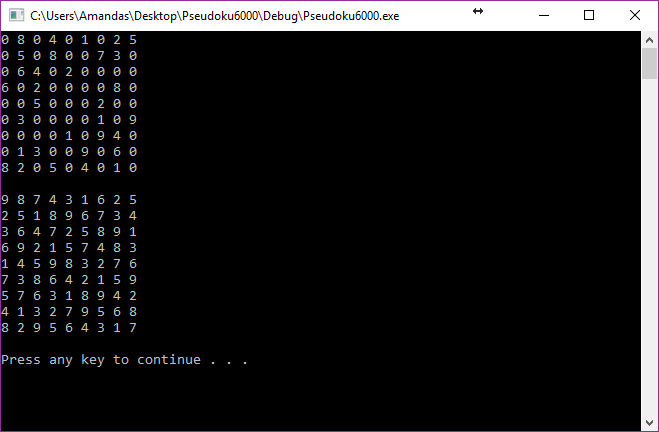
0 3 0 0 0 0 1 0 9

0 0 0 0 1 0 9 4 0

0 1 3 0 0 9 0 6 0

8 2 0 5 0 4 0 1 0

OUTPUT (to console) :



INPUT (from file) :

2 0 4 0 0 1 9 7 3

1 0 3 0 9 0 0 0 0

9 5 0 0 0 0 0 0 0

0 0 9 0 3 7 1 8 0

0 0 7 1 0 2 3 0 0

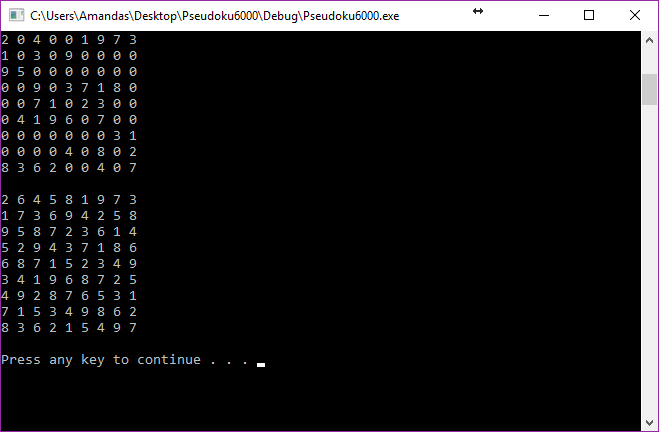
0 4 1 9 6 0 7 0 0

0 0 0 0 0 0 0 3 1

0 0 0 0 4 0 8 0 2

8 3 6 2 0 0 4 0 7

OUTPUT (to console) :



INPUT (from file) :

5 2 0 8 0 0 3 9 0

6 0 9 0 0 5 0 0 0

0 0 0 0 6 9 5 8 0

0 0 0 0 7 0 6 4 9

0 6 0 9 0 2 0 5 0

3 9 5 0 4 0 0 0 0

0 4 2 3 9 0 0 0 0

0 0 0 1 0 0 9 0 8

0 3 1 0 0 8 0 6 2

OUTPUT (to console) :

