Noah Prince

CS2500 – Algorithms

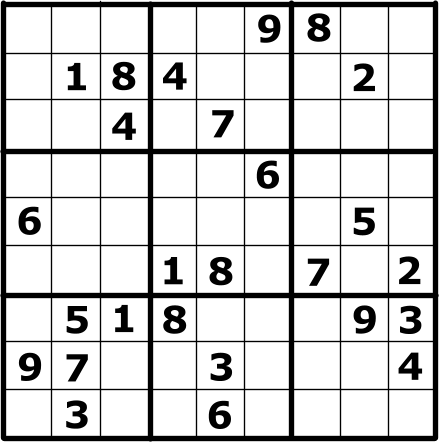
Zhishan Guo

December 8, 2016

**Problem statement**: Sudoku is a popular logic-based combinatorial number puzzle. With a n2 x n2 grid containing n x n blocks, this puzzle is considered NP-complete. For n = 3 (classical Sudoku), the grid becomes a 9 x 9 grid containing a total of 81 blocks.

Classic Sudoku puzzles are defined by three constraints: every row in the puzzle should have unique values (1,2…,9), every column in the puzzle should have unique values (1,2,…,9), and every 3 x 3 box within the grid should contain unique values (1,2…,9). Can Sudoku puzzles be solved in an efficient manner?

**Methods:**

A puzzle is typically given with a given number of the squares already filled in. There are multiple ways to solve a Sudoku puzzle including but not limited to: backtracking, exact cover, brute-force, stochastic search, and constraint programming.

In the puzzle to the left, there are 26 numbers given with 55 blank spaces remaining. A common pencil-and-paper algorithm is to fill in squares with any possible value that they have and from that point on cancel out some of them using the constraints. This same algorithm can be applied with the use of a computer in the case of the Brute-force method by filling the entire board with values and checking for conflicts. If a conflict is found, then continue to fill the board with values until a solution is found with no conflicts. This will always find a solution but the time it takes to find such a solution can be deterministic and primarily come down to whenever you are lucky enough to randomize the correct solution.

In the same puzzle, a stochastic search algorithm can be applied to find the solution. A stochastic search algorithms applied to a Sudoku puzzle would start with an initially random solution to the puzzle, just like the brute-force method. Instead of regenerating another randomized set, the search will calculate the number of errors based on the current grid, and shuffle the numbers in this solution until the error percentage is at a minimum. When that minimum is reached, it will only randomize the numbers in the grid that are still consider wrong.

**My Attempts:** Firstly, I decided on the backtracking algorithm as an implementation because of its relation to depth first search and breadth first search algorithms that we discussed in class. My algorithm is successfully able to determine a solution for any given Sudoku puzzle if a solution exists. My algorithm accomplishes this by using a 2-Deminsional array of integers that become filled with the preexisting board numbers as inputted by a user. A recursive algorithm is then used to determine the solution to the puzzle.

**Pseudocode for input:**

Ask the user for the numbers in each row and represent blank spaces with 0’s

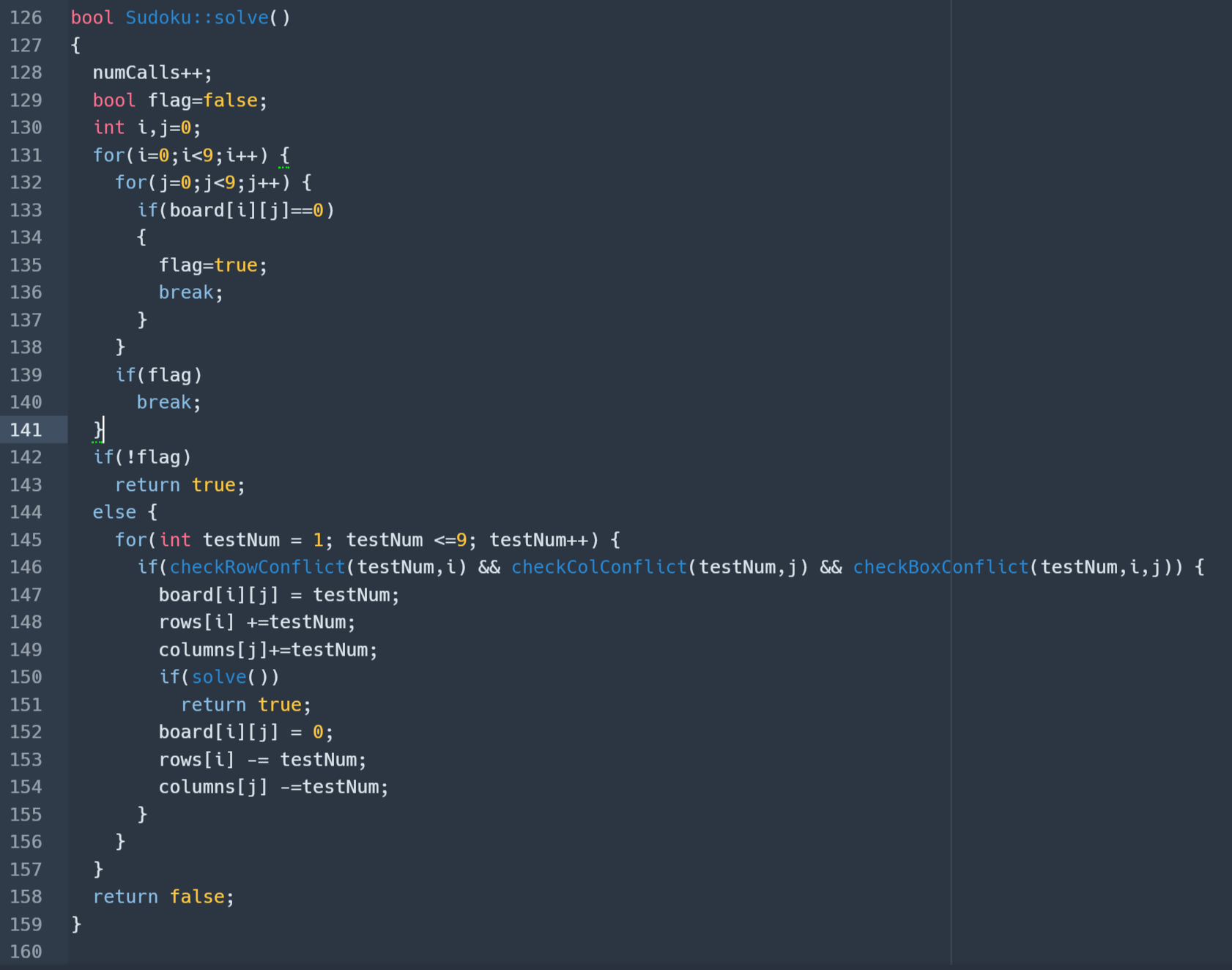
Insert a number (initially 1). Check if there is a conflict in the row, column, or box.

If there is a conflict, increment number. If there is not

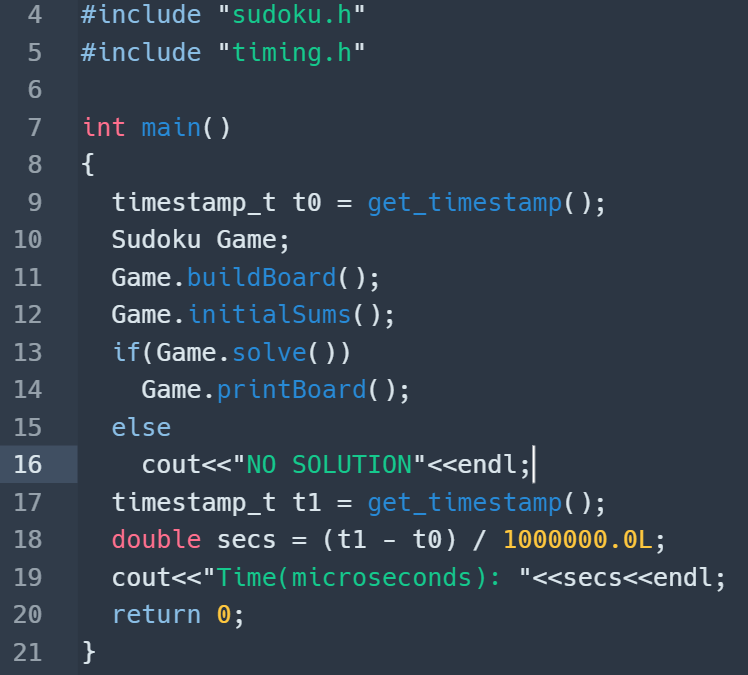
Move to next blank space and insert again. Check for conflicts.

Repeats till solved or back track and increment number.

**Code for algorithm:**

**­­**

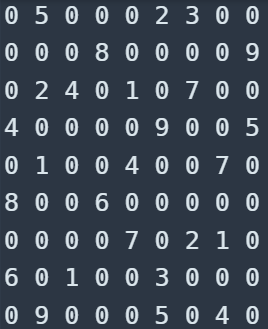
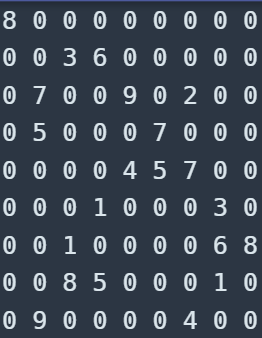
**Code for main:**

****

**Additions:** Instead of just implementing an algorithm we learned in class to a application, I attempted to further optimize it’s utilization by combining the common backtracking algorithm with Constraint Programming techniques. In Sudoku, the rules state that every row and every column will have the numbers 1 through 9 existing in them. By that logic, every row and every column will each add to 45[1+2+3+4+5+6+7+8+9=45]. Adding this constraint to the row and column checks, the algorithm can skip scanning the array multiple times throughout the program.

**Results:**

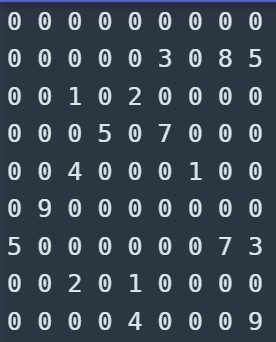
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Boards | Time w/o optimizations  (Microseconds) | Time w/ optimizations  (microseconds) | Number of row scans called/skipped | Number of column scans called/skipped |
| 1 | 40407.3 | 40242 | 49559/37586 | 49559/5271 |
| 2 | 2488.7 | 2268.9 | 611/551 | 611/5 |

 Board 1 Board 2

**Analysis:** Although the time it takes for the algorithm to find the best solution for the puzzle are in Microseconds (which translate to 40 milliseconds and 2 milliseconds respectively), the optimizations save time. What is more noticeable about the optimization is the amount of times a row within the array does not need to be scanned. In the case of Board 1, out of the 49559 recursive calls by the algorithm, it skipped 42857 array scans total. Each array scan would run through a for loop from 1 to 9 and check if the testing number conflicted in the row, but with the added constraint that every row and column could be no more than 45 when summed, there was a storage array for the row and column totals. This allowed the for loop to scan for conflicts to be skipped if the test number would make the row or column total surpass 45. By skipping a for loop of count 9 a total of 49559 times, there were 403,174 less comparisons (9 \* 49559 – 42857). There is minimal time difference, but far less calculations need to be made.

**Runtime: T**he Big-O complexity of this algorithm is O(n^(n^2-k)) where n is the boards width and k is the number of provided numbers at the start. By this definition, the worst-case scenario for a puzzle is one with no given hints. But this is still solved relatively quickly because the board then lacks constraints at the beginning and defines the first row as (1,2,3,4,5,6,7,8,9) and then solves the puzzle. The true worst case is a puzzle with enough hints that makes the first block (1x1) to be 9 and for the algorithm to discover it after scanning the grid a total of 8 times. This is a very difficult situation to come across because there has yet to be a grid found that is this case. In my search for the longest runtime for my algorithm, I found a puzzle that gave my algorithm the most challenge. It took ~26 seconds to find the correct solution with a total of 69,175,317 recursion calls. This is still far less than the worst-case scenario or the worse time complexity of a puzzle with 17 hints.

**Extreme case puzzle**



Even with the large amount of recursion calls, my optimizations were able to shave off 58,570,922 comparisons(roughly 85% of the total comparisons were skipped) or 1 second of the runtime.