Zachary Andrews, Brianna Crawford, Tyler Lyle, Joshua Yee
Computer Science 102 Final Project Report
12/10/18
Bonham-Carter

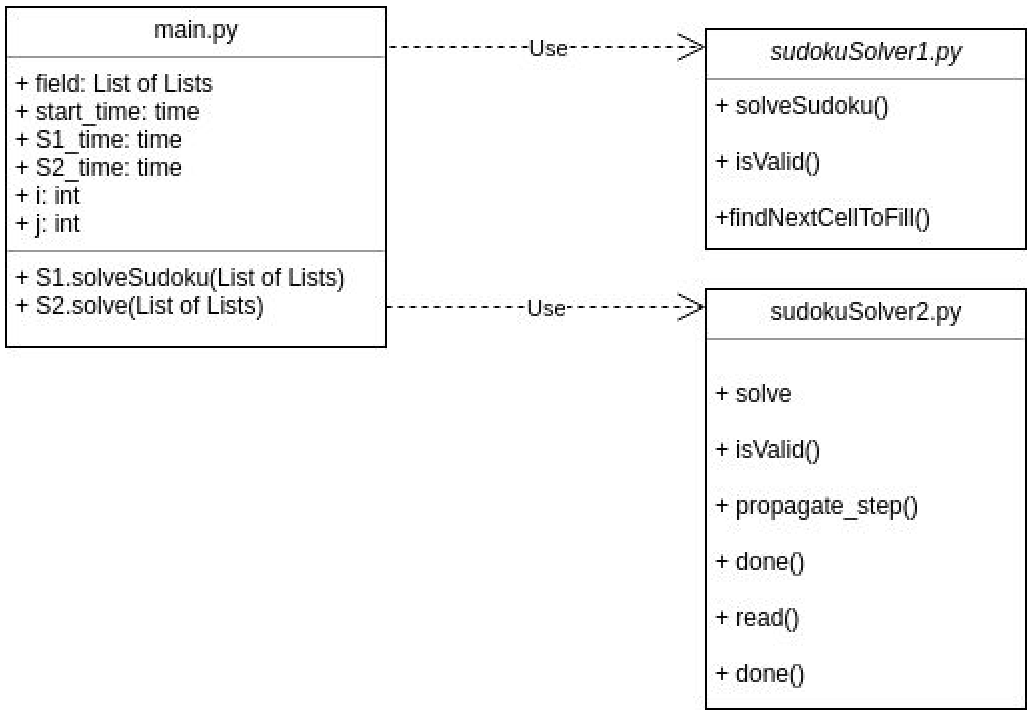
Source of code:
https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution

For our project, we chose to examine two distinct algorithms for solving sudoku puzzles. The first algorithm uses a brute force approach, while the second algorithm takes on a more analytical approach. While we initially thought it would be interesting and fun to work with sudoku puzzles, this project can further be useful when looking at the real-world application of scheduling. Similarly to a Sudoku puzzle where each row, column, and three by three sub-region should contain the unique numbers one through nine only once, when scheduling workers, classes, etc., there should be no overlap where a person would have to be in two different places at once. Therefore, it can be useful when considering time and efficiency to take a more methodical approach to scheduling similar to the second algorithm we analyze, as opposed to simply attempting a brute force approach. Beyond the relation to scheduling, sudoku puzzles are also related to the course topic of discrete structures through graph theory. Sudoku puzzles can be represented as graphs where each of the 81 cells of the sudoku board acts as vertices. Two vertices are connected by an edge if the cells that they correspond to are in the same column, row, or three by three sub-region. Additionally, the numbers one through nine can be assigned to separate colors so that the color of each vertex corresponds to the color of the given number for that vertex/cell.

Each of the two algorithms we utilized had very different structures. The simple brute force algorithm (S1) utilized three different methods. The first method that is called is solveSudoku which we pass the initial unsolved sudoku board into. From there we then determine the next cell that has to be filled, if there are no more cells to fill (and the finished board is a valid solution to the sudoku problem) then the program returns true. If the board is not solved then the next method isValid is called which checks the entire board, both by rows and by columns. Assuming that everything in the board is valid the solveSudoku method is then recalled with the current cell indexes passed as arguments which will be used to determine the next cell to fill. However, if the board is not valid then the algorithm will backtrack until it finds another valid move to make. This process repeats until a solution is found and the condition mentioned earlier is true.

The more analytical algorithm (S2) utilized five methods. The first method that is called is solve, and is the main driver for the algorithm. The board is then processed by the method read which returns the processed board. After this, we call the propagate method which tries to fill in each row, column, and 3x3 block. While trying to do each of those actions the row/column/block is checked to ensure that it is always a set and does not have any repeating values. After performing this action it is determined if there will be a solution and a true or false value is returned. If it is determined to not be solvable then None is returned. If that passes then it then checks to see if the board is solved by using the done method which again checks each row and column and checks to make sure they are all sets and returns true or false depending on that condition. If it is not done then the code will recall the solve method and try again using a slightly different initial board. This process like with the brute force method will repeat until there is a solution found.

Finally, we have our main file which we create the board in and call the methods in each algorithm to solve the board. We also keep track of the total runtime for each algorithm within this class to ensure that the runtimes are as accurate as possible. The interactions between the main file and our algorithms can be seen in the UML diagrams below.



The results were very interesting in terms of the time complexities and the overall use of code that went into making the two different sudoku solvers. The first one, which we call S1, takes a brute force method. However, the code is short and sweet in terms of readability and literal lines of code. The second method, which we call S2, takes a more complicated approach with specific conditions. Initially, we assumed that the code of S1 would be faster because there were not as many moving pieces running through the program. However, through seeing the time difference of the programs through solving the puzzle once, we can see that the algorithm of S2 is actually roughly 100 times faster than the S1 algorithm. Knowing this, we increased the number of times that it solved the puzzle exponentially and the results were also very interesting. The puzzles that the algorithm needs to solve, the longer S2 took to complete the puzzle. By 1000 puzzles, S2 became 5 times slower and from 10,000 to 1,000,000 S2 became 10 times slower than the S1 algorithm. This is likely due to the fact that S2 is much more specific and complex in design which is what we attributed this to.

However upon further inspection of the code and how it runs, we found that S2 was, faster because it is more specific so it is able to more reasonably run through the data and help to create a fast easy way to check the numbers as it goes. Unlike S1 that creates a brute force approach which means it has to run through iterations multiple time which creates an exponential increase in the runtime of the program. Running the data again with the new implementation, we can see that 100 times on both, we saw that the time it took for both was significantly larger with S1 taking up to 15 seconds and S2 taking 2 seconds. However, this is still a significant amount of time saved as opposed to S1.

In conclusion, we have chosen these two classes because of their vast difference in approach to solve the same problem. We looked into the time complexity of both sets of code and we discovered that they both have the same time complexity. Knowing this we looked more into how the program functions and with this, we have discovered the usefulness of both a straightforward method of a solving algorithm, as well as a specific check that calls on multiple methods. Ultimately the algorithm of S2 which is much more complex in design will be able to run more checks to keep the function moving quickly. Whereas S1, which was much more simple, ended up being much slower under these same conditions. Knowing this we can likely apply this same idea to more similar algorithms in order to find the fastest algorithm based on the user's needs because though S1 probably took less time to create, S2 is much more effective overall.

Joshua helped to find the code used in this project and worked on writing the results and the conclusion for this paper. During the presentation, he worked on the results and demo of the code. Something that all of us had worked on together was analyzing the different outputs of the time complexities and the results of the runtime for both of the codes used.

Zach also helped find the code used for the project and modified it to line up with what we needed to do. For the presentation, he analyzed the two different algorithms in order to see why they behaved the way they did and present the findings to the class. He also discovered issues with our implementation of the code which caused our results to be skewed and not line up with what we predicted.

Brianna worked on the motivation for both the paper and the presentation.

Tyler helped find the code used for the project as well as the idea to create a sudoku solving bot in the first place. For the presentation specifically, he worked on the conclusion and results of the slide. And for the paper, Tyler worked on the UML diagram. We all worked on the code and results.