

**ECSE 420 GROUP PROJECT**

Sudoku Solving Algorithms

Project Group 11

Members:

Batin Can Balkan - 260725650

Erion Hysa - 260746699

Aleks Murauskas -260718389

Nicolas Barreyro - 260730549

**Introduction**

To gain more understanding on how, and what our algorithms do, one must know what Sudoku is and how it is solved. Sudoku is a logic based number placement puzzle. The objective is to fill a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 subgrids that compose the grid contain all of the digits from 1 to 9. Figure one shows what a solved sudoku puzzle might look like.

Firstly, all our algorithms must obey the general rules of Sudoku. Which can be summarized as the following:

i) Each cell must hold a value between 1-9

ii) Each row, column, and block can have no repetitive values.

iii) Smallest number of clues is seventeen.

In addition to that, we have decided to write algorithms that will be able to solve Sudoku puzzles in a limited amount of time. We have used four different approaches: First, we have coded a brute force algorithm sequentially. Then, we parallelized that algorithm to get better results. Finally, we have implemented Crook’s algorithm both sequentially and parallelly to solve the puzzles. In the latter section, for each algorithm we have implemented we explain how to run and compile the program, problems addressed by that particular algorithm, the code structure we have used, algorithm analysis, advantages/disadvantages, and finally our test approach/ results.

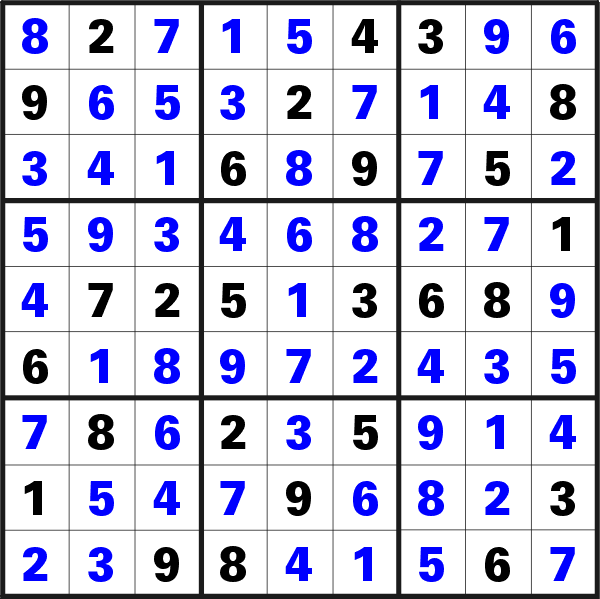
****

Fig 1. Example of a solved Sudoku puzzle

**Algorithms**

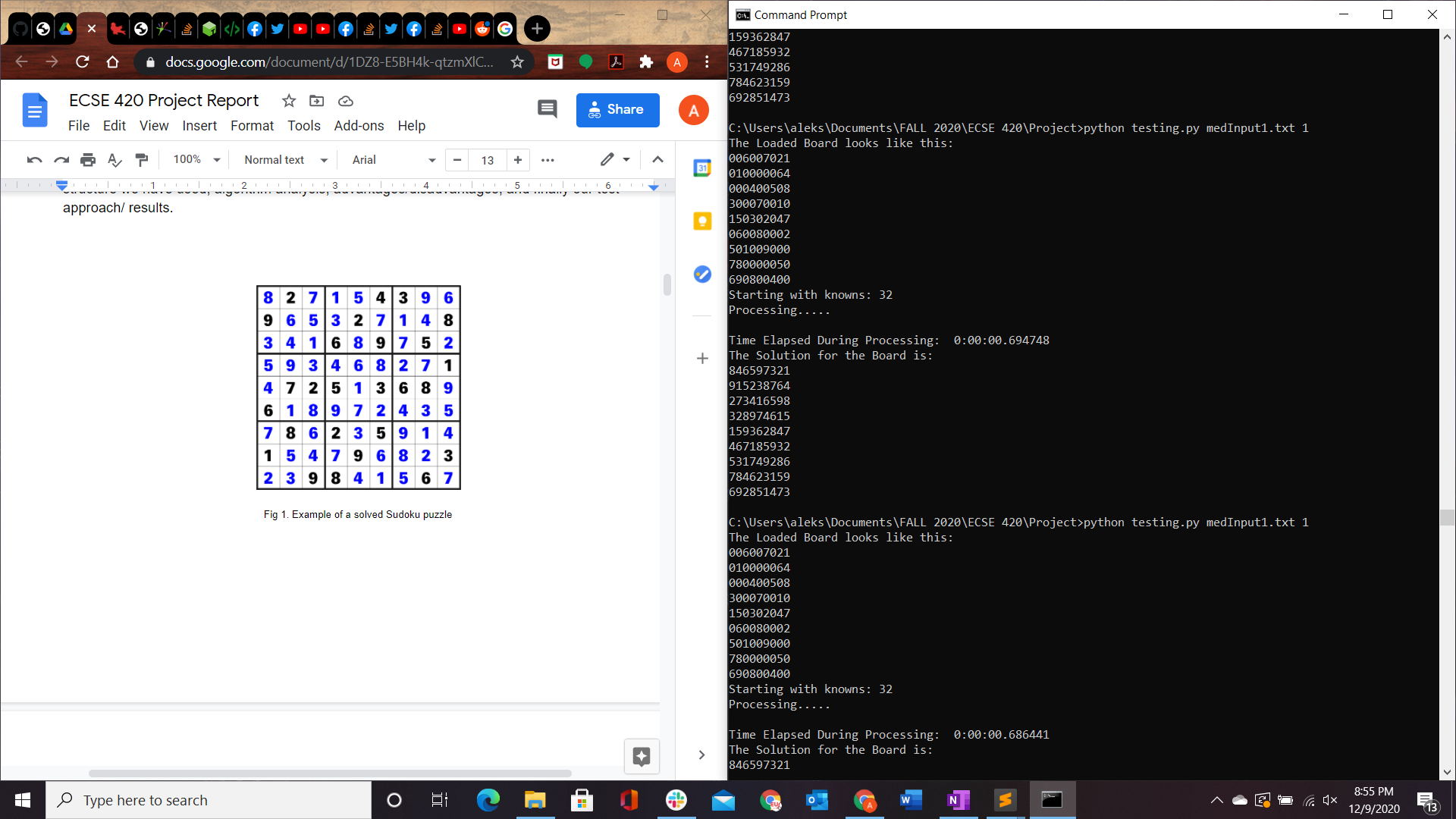
**1) Sequential Brute Force**

**i) Preparation:**

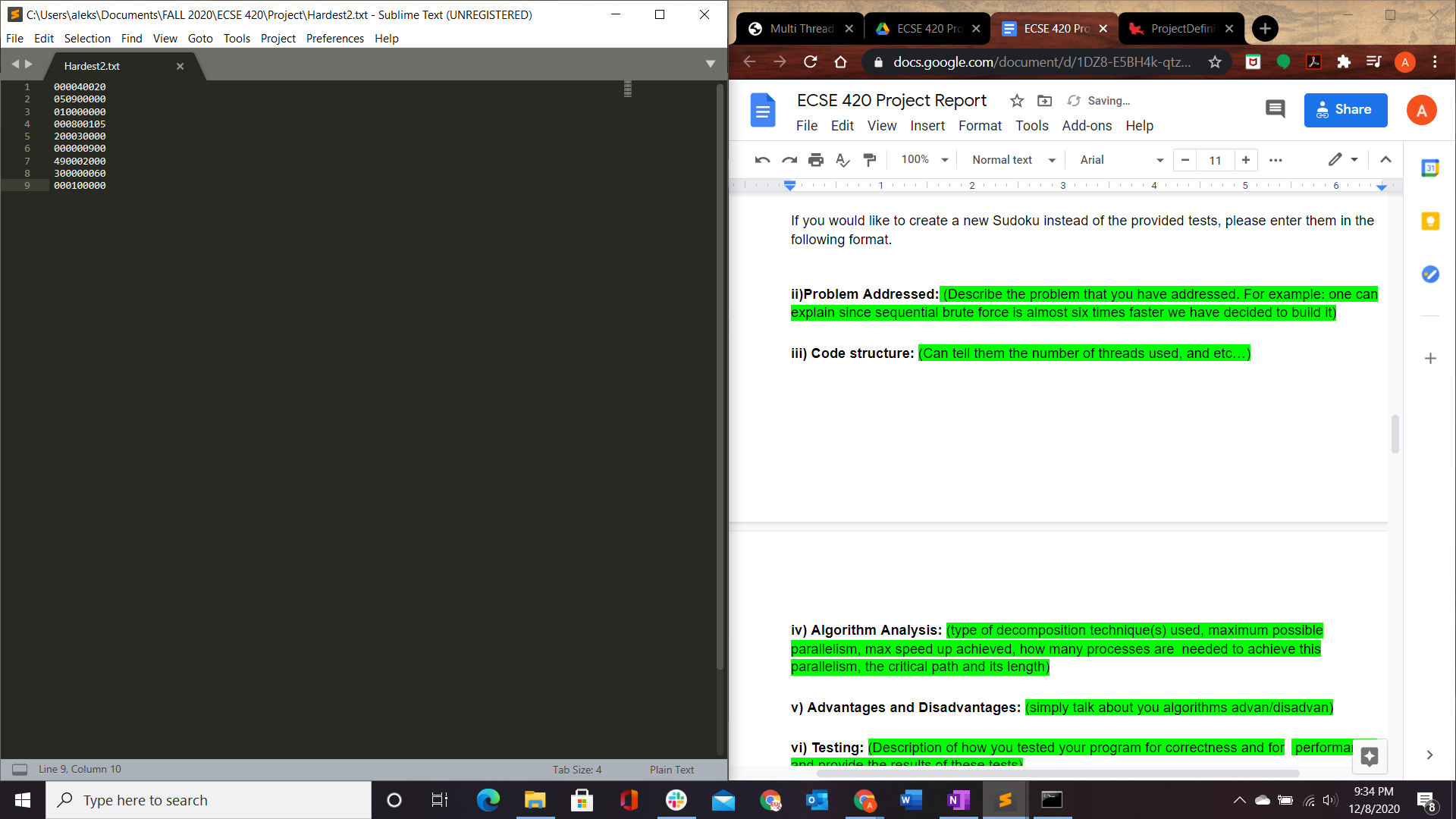
Our Sequential brute force algorithm is held in the file BruteForceSudokuSolver.py.

In order to solve a Sudoku, place the BruteForceSudokuSolver.py in the same folder with the desired test input. Change directory until you are within this file, then input the following into the command line.

>python bruteforceSudokuSolver.py [Input File Name] 1

An example of an input line and the expected output. 

If you would like to create a new Sudoku instead of the provided tests, please enter them in the following format.



**ii)Problem Addressed:**

This algorithm can solve any solvable sudoku puzzle, and generally in an efficient time frame. However in the case of very difficult puzzles, the solver may take several minutes to find a solution.

We had difficulty turning our original brute force algorithm into a parallel program, It is submitted as OriginalBruteForce. It has better sequential execution times, but we were unable to create a parallel version of that algorithm without clashing race conditions or infinite loops.

**iii) Code structure:**

Since this is a sequential program, only one process is used.

The processing begins in main, which uses sys to retrieve the argument file and number of processors, in this case 1. If the number of processors is less than 1 or greater than 32, the program will terminate.

The method loadBoard() takes the input file as an argument and prints that board to console as well as it then parses the string into a list of list, representing a 9x9 board.

The method Algorithm() intakes the starting board and a number of processors. The method creates a copy of the given board and counts the number of empty cells. If this board has no empty cells, it is already solved and terminates. a Pool of the given number of processors is created. Within an infinite loop, a list of all possible boards is created, then a starmap runs board\_verification() on all the possible tests. This is the only portion of the code that is parallelized. Then the list of tests are parsed, and the successful tests are reintegrated into the list of possible solutions, and the loop iterates.

The method board\_verification() takes in two 9x9 boards, the given board and the solution. The method will iterate over every empty cell, and then check if that value breaks the row column or block sudoku rules.

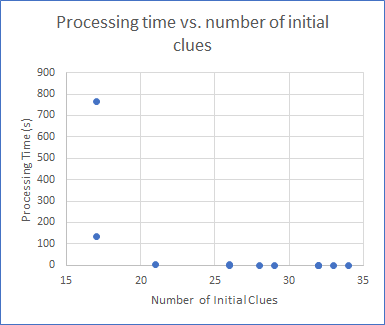
Once Algorithm() returns a puzzle, that is the solution, but if it runs out of possible solutions before then, then it is unable to find a solution.

**iv) Algorithm Analysis:**

This program is entirely sequential, we will discuss algorithm analysis within the next section.

**v) Advantages and Disadvantages:**

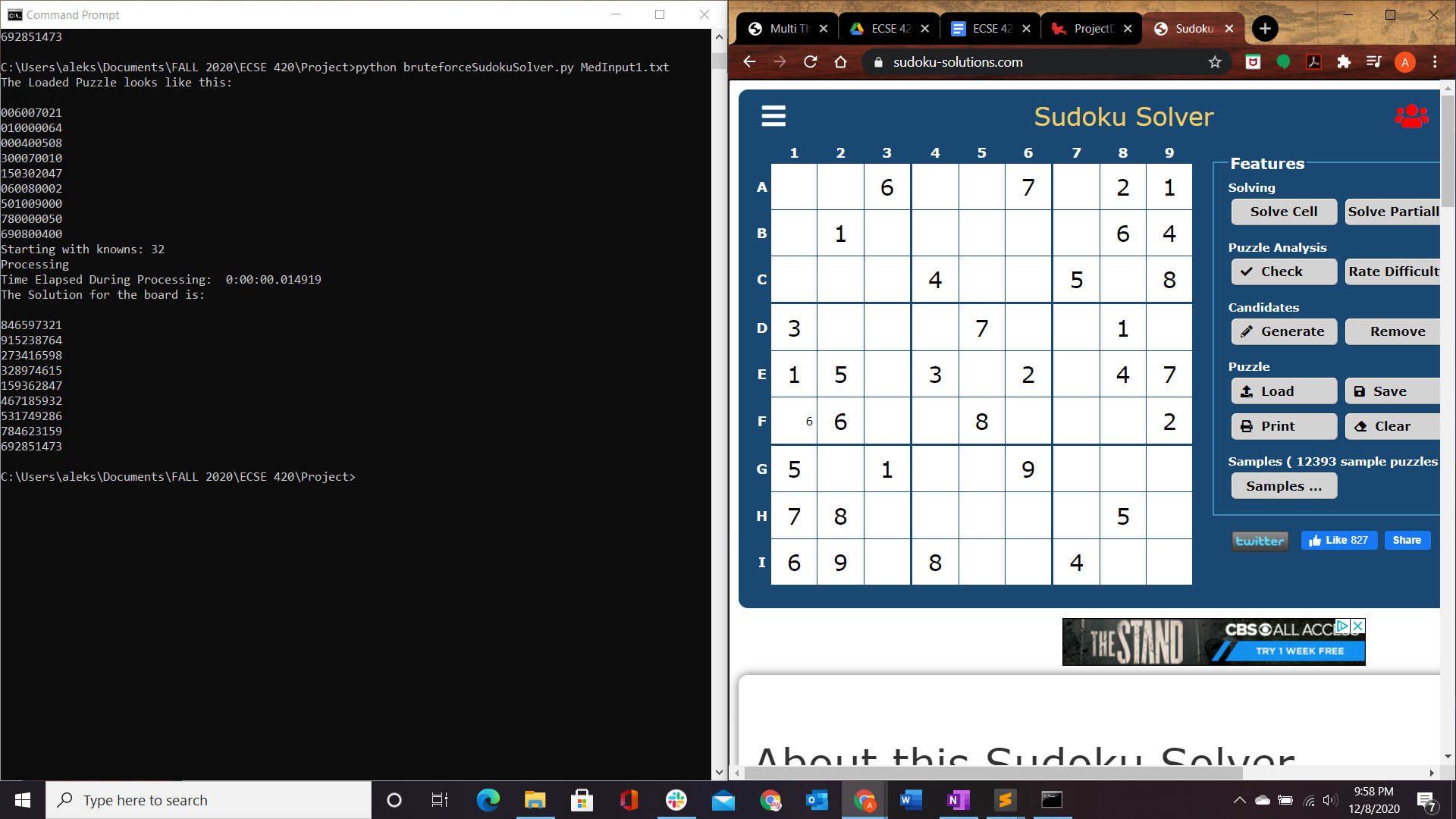
This method will always find a solution if one exists, however as it does not use parallelism it can take quite a long time to execute. We found the execution time was not reliant on the given difficulty, but rather execution time scaled heavily with the number of given knowns at the beginning.



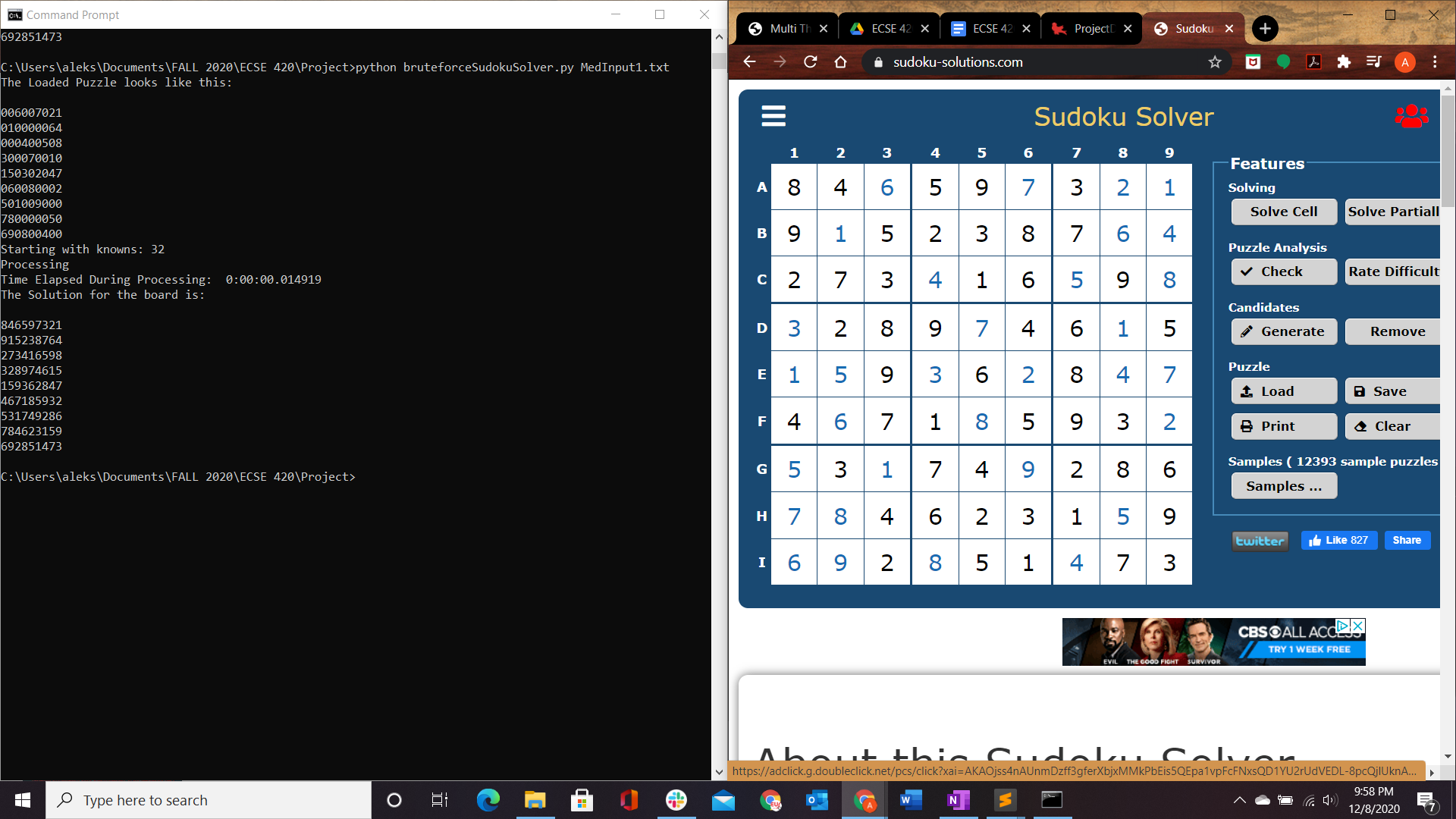
**vi) Testing:**

We tested performance by setting recording the time before execution, and then again, after the solution was found, and printed the difference to the console. We found that, after using several boards of

We tested correctness by comparing the input and output boards to the solutions given by <https://www.sudoku-solutions.com/> on the same input boards.

****

Example of comparison of input boards.

****

Example of comparison of output boards after solutions found independently.

**2) Parallel Brute Force**

**i) Preparation:**

Our Sequential brute force algorithm is held in the file BruteForceSudokuSolver.py.

In order to solve a Sudoku, place the BruteForceSudokuSolver.py in the same folder with the desired test input. Change directory until you are within this file, then input the following into the command line.

>python bruteforceSudokuSolver.py [Input File Name] [Number of processes]

The expected output is identical to that of the sequential Brute Force.

**ii)Problem Addressed:**

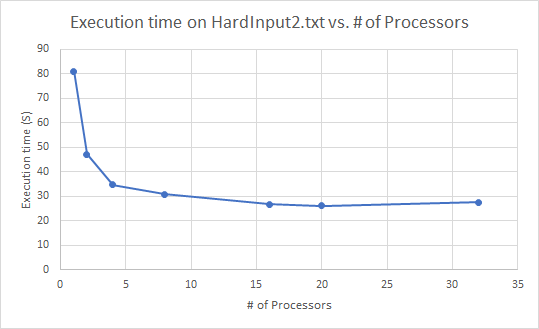
We created this algorithm to increase the execution speed of our brute force algorithm and see if this algorithm achieves a higher speedup than that of crooks, in order to determine the Ideal Algorithm for the solution of a Sudoku board.

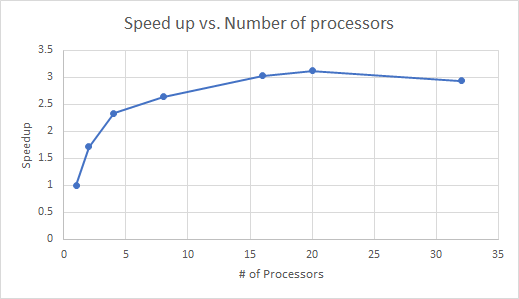
**iii) Code structure:**

The Code structure can be seen in the previous section. The only change is that board\_verification() occurs on multiple processors at once. We allow between 1 and 32 processors, as any more consistently resulted in busy waiting and increased execution time.

**iv) Algorithm Analysis:**

The Maximum speedup of 3 is achieved at about 16 processors. In order to further increase the speedup, a larger proportion of the code could be parallelized. The creation and revision of possible solutions, at the beginning and end of each loop could be parallelized to further increase speedup, as that is the primary portion of overhead.



****

**v) Advantages and Disadvantages:**

In cases where the sequential time is long, more than one minute, a speed up of about 3 can be achieved. However in cases where the sequential was incredibly fast with a execution time of less than second, using parallelism does not increase execution speed in any significant way, and in fact due to the busy waiting processes, increased execution time.

**vi) Testing:**

Correctness was tested by inputting the initial boards to <https://www.sudoku-solutions.com/> and comparing the answers. For boards that had multiple solutions, the values for all the numbers from the solution were input to the website to validate the final output of the program.

**3) Sequential Crook’s Algorithm**

**i) Preparation:**

Our sequential crook’s algorithm implementation is contained in the “main.py” folder under the “sequential\_crooks” directory. Along with this file, we have several unsolved sudoku boards ranking from easy to very hard difficulty. In order to test the algorithm, the filename of the sudoku to solve can be set.

**ii)Problem Addressed:**

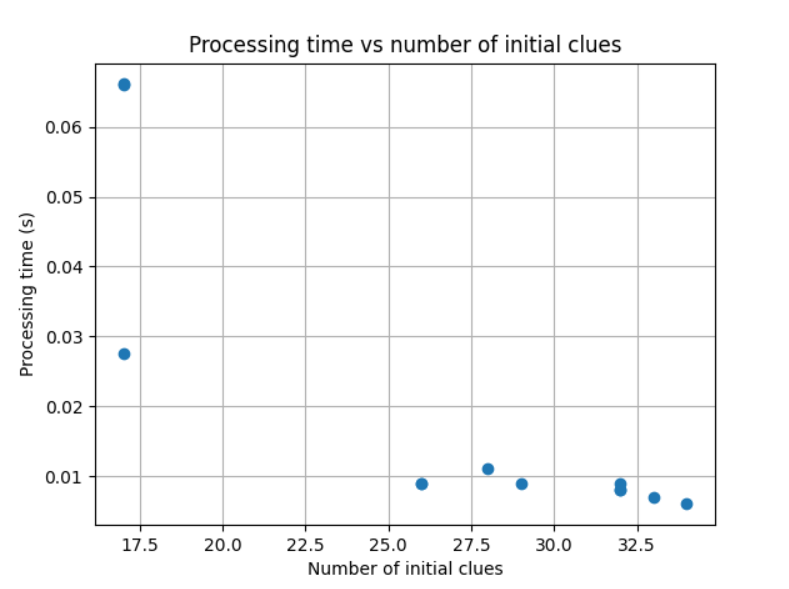
Brute forcing a sudoku does find a solution, however the time taken is not optimal. Using Crook’s algorithm, we can implement a method of logic to prioritize setting numbers to optimal cells and obtaining a final solution in less steps. This will improve efficiency and remove some of the redundant moves that would be performed by a brute forcing method.

**iii) Code structure:**

Given that this is a sequential implementation of the Crook’s algorithm, we have only one thread running. The code is organized into the main function and two groups of helper functions: one that corresponds to the basic rules of sudoku (i.e checking that there are no duplicate values across a row/column/box) and one that corresponds to the rules of Crooks. These rules will be covered more in detail in the next subsection and the parallel Crook’s implementation section. The check for the basic rule of sudoku is applied every time we update the board to make sure that we are not allowing duplicates and the rules for Crook’s algorithm are applied continuously as well.

**iv) Algorithm Analysis:**

This program is entirely sequential. The parallel form of Crook’s Algorithm will be discussed in the next section. We found that this algorithm scales similarly to Brute Force, however with a much smaller base time, regardless of difficulty.



**v) Advantages and Disadvantages:**

We obtain a considerable speedup compared to the sequential brute force algorithm as can be seen on the graph of Processing time vs number of initial clues, especially for the easier sudoku boards. This scaling with a number of clues follows a roughly identical trend to that of Brute Force. We now know that Crook’s will always be faster than Brute force as long as it can find a solution.

**vi) Testing:**

Correctness was tested by inputting the initial boards to <https://www.sudoku-solutions.com/> and comparing the answers. For boards that had multiple solutions, the values for all the numbers from the solution were input to the website to validate the final output of the program.

**4)Parallel Crook’s Algorithm**

**i) Preparation:**

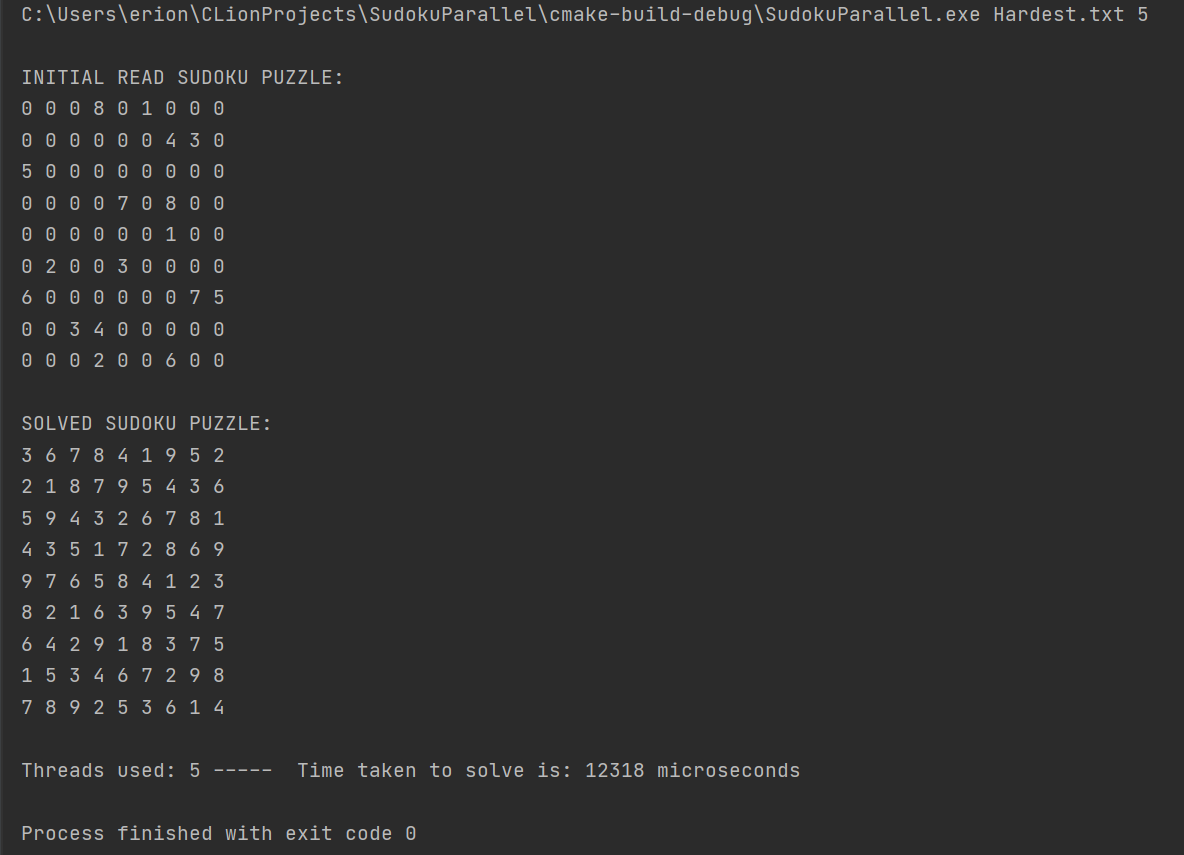
The file that contains the code to run parallel Crook’s is called parallelCrooks.cpp. To compile it one can use a g++ compiler from the command line using the command once in the file’s directory:

$ g++ -o parallelCrooks parallelCrooks.cpp

This will compile the program parallelCrooks which must be run with two arguments from the command line. The first argument has to be a .txt file containing the sudoku puzzle under the same format as in sequential brute force. This .txt file must be in the same directory as the compiled program. The second argument must be the number of threads to use. The program can be run with the command:

$ ./parallelCrooks file\_name thread\_number

An example of an input line and the expected output:



**ii)Problem Addressed:**

Sequential Crook’s algorithm is already quite efficient. The issue was that Crook’s algorithm does not always find the solution. There must be some sort of backtracking to solve more complex puzzles when it happens. By using a backtracking algorithm that is threaded we can parallelize the part which takes the longest and help speed up the amount of time taken when Crook’s algorithm stops because it is blocked and can not take a guess.

**iii) Code structure:**

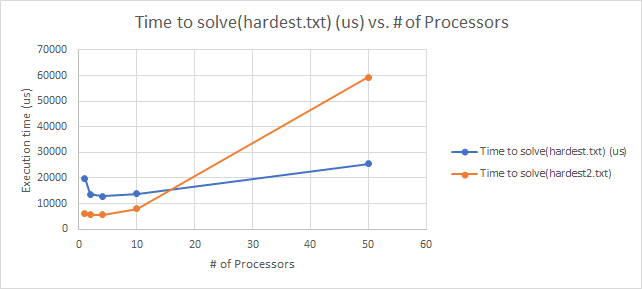
Once the code gets the input for the filename it reads and populates an array of 9x9 which serves as the sudoku board. This is all kept track of in the sudoku class which holds elements and methods used during the algorithm. The puzzle is initially marked up and then the Crook’s part of the algorithm is run. First the elimination part is run to check if there are cells with one possible value or not to update. Otherwise we check for singletons by looking if there is one possible value that is possible in a combination of rows, columns and boxes based on the sudoku rules. The third part of Crook’s is to find preemptive sets which “is composed of numbers from the set [1, 2, . . . , 9] and is a set of size m, 2 ≤ m ≤ 9, whose numbers are potential occupants of m cells exclusively, where exclusively means that no other numbers in the set [1, 2, . . . , 9] other than the members of the preemptive set are potential occupants of those m cells” [1]. If at any steps there are changes we restart from step one and make the way down again. If the Crook’s is stuck we backtrack.

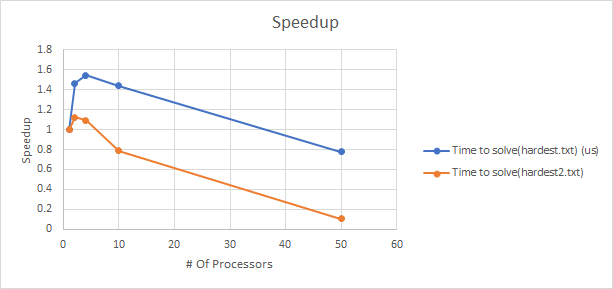
In backtracking the set thread number from the command is used to create the threads. And each thread has its own stack and starting board. Each thread fills spots in the board and then puts back the valid sudoku boards in its own stack. It then takes another puzzle board state and repeats. Since the stacks belong to each individual thread there is less of an issue to manage one global stack. The threads are then all merged and we are left with one final board once the backtracking algorithm terminates with the output of the solution.

**iv) Algorithm Analysis:**

The inputs and puzzles used were of varying complexity. The easy, medium and hard ones used did not make the crook’s algorithm block. In fact the threads did not matter for these puzzles as they were solved in 30-40 microseconds for easy. 50-60 microseconds for medium and 60-80 microseconds on hard. The inputs that were interesting for this part are the hardest.txt, hardest2.txt. In these puzzles the crook’s would be stuck after a certain point leaving it to the backtracking algorithm. Here are the results for the time to solve the hardest puzzles.

|  |  |  |
| --- | --- | --- |
| Number of threads | Time to solve(hardest.txt) | Time to solve(hardest2.txt) |
| 1 | 19856μs | 6257μs |
| 2 | 13569 μs | 5568μs |
| 4 | 12854 μs | 5716μs |
| 10 | 13814μs | 7922μs |
| 50 | 25508 μs | 59364μs |





We can see that the time to solve the two different hard puzzles is affected by the number of threads chosen. With 1 thread there is still a solution that is found however it is not the fastest. It is important to note that multiple threads does not result in a linear speedup. As can be seen from the data the time to solve speeds up by 1.2 to 1.5 from 1 to 2 threads depending on the puzzle. However, this speedup does not change much more once we increase the threads from then on. In fact at 10 threads we notice a decrease in performance already with the second puzzle having a worse solve time than with one thread. This is probably due to the constrained size of the board of 9x9 and the fact that sometimes a larger number of threads will slow down the program rather than speed it up. The increased number of threads seems to have resulted in busy waiting.

**v) Advantages and Disadvantages:**

The advantage of this solution is that it is quick, efficient and so far has been able to find solutions for the puzzle. The fact that each thread has its own stack rather than a shared one also helps with managing resources. However, this could come with a price since maybe one thread could be overloaded and have much more boards to process relative to the other threads, making it so that the efficiency will be reduced since it can be a bottleneck. Additionally having a larger number of threads tends to slow down the backtracking relative to a lower number. Finally, the more progress that the Crook’s portion of the algorithm performs the less the performance will be affected by the threads for the rest of the backtracking.

**vi) Testing:**

Correctness was tested by inputting the initial boards to <https://www.sudoku-solutions.com/> and comparing the answers. For boards that had multiple solutions, the values for all the numbers from the solution were input to the website to validate the final output of the program.

**Conclusion**

In conclusion, we found that Crooks algorithm consistently solved the Sudoku puzzle faster than the Brute Force Algorithm. However, the Brute Force algorithm scaled better with parallelization than the Crooks. Brute Force was able to achieve a speed up of 3, but only on puzzles where the sequential brute force had a long execution time. In cases where the sequential was fast, there was little speed up however.

In the case of the Crook’s Algorithm, Crook’s consistently solved the puzzles faster than brute force. However, Crook’s does not parallelize as well, only achieving a maximum speedup of 1.2 to 1.5 at about 2-4 processors. Once the number of processors exceeds this, Busy waiting becomes a significant issue, increasing execution time over the sequential time.

In both cases, the longer the sequential process took, the more effective the speedup was before the point of diminishing returns.

The ideal way to solve a Sudoku puzzle with our algorithms is this:

1. Feed your puzzle into the Crooks algorithm with 4 parallel processors
2. If a solution is returned, it is solved
3. Otherwise, Crooks cannot solve the puzzle, feed it’s result to brute force with 16 processors
4. If a solution is returned, it is solved
5. Otherwise, The given puzzle is not solvable

**References**

[1] Crook, J. F.. “A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles.” (2009). [Online]. Available: <http://www.ams.org/notices/200904/tx090400460p.pdf>. [Accessed 01-Dec-2020]