A2 Report

Setup

I have 3 classes in SudokuSolver.py. One for GUI, one for Command Line instance and one is the actual puzzle with solvers. The actual Puzzle has list of rows (lists) that stores the Puzzle (i.e. 9x9).

I have made the GUI so that each node visit updates the cell value in the GUI as well. To run the algorithm faster, you can uncheck 'Update GUI' option. I have also added file selector to select the file to run and a dropdown to select the algorithm to run. I have also put labels to display node count, total time and algorithm runtime. At first I was changing the background colors of the cells that are being updated to display the process better but it slowed down the performance by a lot. Thus I went back to just updating cell values.

Note: For checking runtime, command line version is recommended
To run the GUI version:
```
>>> python2.7 SudokuSolver.py
```
I use command line class if there have been arguments provided.

Algorithm Comparisons:

Brute-force	puzzle1.txt	puzzle2.txt	puzzle3.txt	puzzle4.txt	puzzle5.txt
Nodes Explored	N/A	N/A	N/A	N/A	N/A
Total runtime	N/A	N/A	N/A	N/A	N/A
Algorithm runtime	N/A	N/A	N/A	N/A	N/A

Backtracking	puzzle1.txt	puzzle2.txt	puzzle3.txt	puzzle4.txt	puzzle5.txt
Nodes Explored	687	8008	14340	4621	2359
Total runtime	101.65	102.46	119.02	115.04	103.02
Algorithm runtime	1.47	12.20	21.81	8.26	7.03

FC with MRV	puzzle1.txt	puzzle2.txt	puzzle3.txt	puzzle4.txt	puzzle5.txt
Nodes Explored	41	81	357	69	66
Total runtime	102.21	102.41	104.42	102.58	102.07
Algorithm runtime	1.61	2.72	10.32	3.46	3.32

1. Brute-force

In Brute-force algorithm, I use recursion to set one value at a time. The domain is values 1 to 9. When we realize that there are no more empty cells, we check to see if the puzzle has been solved, if not we go back and try the next value.

This is of course bad because we don't do any checking to look for what's available and we just try all possible values for all empty cells. This replicates depth-first search with branching factor being 9 as domain length.

This algorithm never ends since there are too many cases to consider for the given test cases. For example, for puzzle1, there are 41 blank spaces. The algorithm will have to go through maximum of $9^{41} = 1.33 \times 10^{39}$ permutations to get to the answer.

n = row/col size (9); **d** = filled out cells; **m** = $n^2 - d$
Time complexity: $O(n^m)$ = **$O(9^m)$**
Since we consider all the cases, each cell can have 1-9 values, therefore the total permutations are 9^m, where m is the number of empty cells.

Space complexity: **O(m)**
Since this is a recursive implementation of Brute-force, the worst case scenario will of course be O(m) where m is the maximum tree depth or number of empty cells in this case.

2. Backtracking

Using Backtracking, we add one more step to the brute-force algorithm. At each step, after adding values to the cell, I check to make sure that the same value is not in the row, column or 3x3 block of the given cell. If it is, we ignore that value and set it back to 0 since it is not a valid move and we move on to explore the next value. And if adding the value at that cell is legal, we set that value and call the function recursively on rest of the empty cells.

This is way faster than Brute-force because by here, we are eliminating all branches of Brute-force that have nodes that clearly do not meet the requirements of a Sudoku board by doing a check at each node. We can see from the table that it finishes in very short time with reasonable number of nodes expanded.

n = row/col size (9); **d** = filled out cells; **m** = $n^2 - d$
Time complexity: $O(n^m)$ = **$O(9^m)$**
The time complexity for worst case in Backtrack is same as Brute-force algorithm since we will be going through maximum all the possible solutions (9^m) to get to the answer.

Space complexity: **O(m)**
As mentioned above, the Space complexity for worst case in Backtrack will also be same as Brute-force since we are we are using recursion and maximum tree depth at given time can be m, the number of empty cells in the grid.

3. Forward-Checking with Minimum Remaining Value Heuristic

In Forward-checking, we initially pre-generate list of possible values at each empty cell in the grid. When we try a value at that point, we remove that value from the same row, column and block and than try to fill in the rest of the cells recursively.

Minimum remaining value heuristic makes this a CSP. It makes sure that we always branch on a node with the smallest remaining values.

So using FC with MRV, we always branch on the cell that has least number of valid values. This gives us more chance of success since it produces skinny trees at the top. This means that more values can be checked with fewer nodes searched, which guarantees more constraints and domain wipeouts with less work.

As we can see in the results, FC with MRV is way faster than Backtrack in terms of speed and in terms of nodes explored. The difference is that forward checking makes sure we know which values are valid beforehand, whereas Backtrack checks all the values on the go. Here, FC with MRV also make sure that we fill cells in the order that gives us minimum chance of failure (i.e. if we have nodes a and b with 4 and 2 valid values in that order, expanding b gives us 50% chance of success compared to 25% of a).

n = row/col size (9); **d** = filled out cells; **m** = $n^2 - d$
Time complexity: $O(n^m)$ = $\mathbf{O(9^m)}$
The worst case or higher bound of time complexity for MRV is also 9^m in this case/ Although we make improvements selecting the most efficient paths by using MRV and eliminating invalid nodes using Forward Checking, worst case remains the same.

Space complexity: **O(m)**
Worst case space complexity for FC-MRV will be O(m) since we will be saving maximum of m nodes while using recursion, where m is the maximum tree depth or number of empty cells in a puzzle.