Description of algorithm

My algorithm uses a depth first search akin to a backtrack search. It does this by choosing the first empty spot in the suduko and for every value, 1-9, places it in that location if it is valid. It then repeats this process. Once a full suduko is found, it must be a solution. Otherwise, if no number can be placed in a given location then the suduko cannot be completed in this path so a bad output or a suduko full of -1's is output.

To assist the simple backtracking algorithm and add optimisations, one part I added was that when it got the list of possible moves/numbers it could play, it always picked the location where the least number of solutions thereby reducing the number of nodes it had to search. We know this because if we take the first location to have 9 possibilities, the next to have 8 etc until 1 possibility, we have 9^2 + 8^2 + 7^2.. +1^2 = 285 nodes whereas if we do the reverse, 1 possibility, 2 possibilities ets we have 1x9 + 2x8 + 3x7… + 9x1 = 165 nodes thereby a 120 node or 42.1% improvement.

My implementation of said algorithm has a speed complexity of O(n!) as each node layer has a multiple of the one before which is alike it being factorial and a storage complexity of O(n) as the depth first search only has to store the current list of nodes that it is currently on.

Optimisations

Prior to said algorithm in the same function, I also have a simple algorithm to place numbers in all squares with only possibility which saves on number of nodes and also allows me to count the number of possibilities for each at the same time.

As well as that in the initialization step, I both validate the suduko so that it actually has a solution and count the clues as there cannot be a sudoku with a unique solution which has less than 17 clues to begin with as is proven by McGuire(2014).

Within the code I also employed some other optimisations to reduce the time of some aspects such as my use of some global variables such as BadOut which contained the value that I would output if there was no solution to a sudoku, the alternative was calling the numpy function every time I wanted to use it which would be slower and more complex.

Reflections and suggestions for further work

One way that the speed of the algorithm could be easily improved would be to rewrite the code in C as it is inherently roughly 10x faster and often C compilers would be smart enough to do their own more in depth optimizations. If using python was still a restriction than another way could be to use more libraries which are written in C.

An algorithmic way of improving the speed of the algorithm could be to implement actual tactics or strategies such as hidden singles, naked doubles and triples Tuleja's theorem and others mentioned by Davis (Shubin, Hayes and Alexanderson 2011 p31-60) which having their roots more deeply in mathematical theory than computational theory may result in some sudoku's being solved significantly faster however the ones that take more node searches may take longer as each individual node in the search may be more complex.

Possibly usage of more complex data structures such as linked lists, arrays or creation of custom data structures and relevant functions for some of the functions may have sped up the program however this is unlikely to significantly increase performance.

References:

Shubin, T., Hayes, D.F. and Alexanderson, G.L., eds, 2011, Expeditions in Mathematics, United States of America: American Mathematical Society

McGuire, G., 2014. There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration. Experimental Mathematics, 23(2), pp 190-217