Readme attempt 2:

For this coursework we were required to implement a sudoku solver.

How I implemented my sudoku solver for the AI coursework:

The solver required representing the sudoku grid with as a state of what number all of its cells were

set to. Then a backtracking depth first search algorithm that was used to test multiple different

possible values for each cell until it either reached the goal state (a complete grid that only contains

one of each number in every row, column and 3x3 square) or worked out that the grid didn’t have a

solution. The algorithm that I used is based off the backtracking algorithm provided in the 8-queens

Jupiter notebooks resource. The idea of the algorithm is that it recursively trials a possible value for

each of the empty cells in an order, then if it reaches an invalid state (defined as being having no

remaining possible values for a cell with a value that is still unknown) it will backtrack up a state and

will try a different possible value for that cell. If the algorithm recursively unwound to the start state

where after trailing every possible value for that tree of states, you could determine that the grid

was had no solution. This meant that you couldn’t cut corners and try make it complete less

recursive calls, because unless it tried all possible values for that set of cells then it might not find

the solution. So instead, to make the algorithm more efficient I implemented a heuristic method that

would choose the next cell to trail a state with based on which had the least number of possible

values. (where ‘possible values’ means: the numbers out of 1-9 that the cell can be set to based on

the numbers that aren’t in that cell’s row, column and 3x3 square). Reason for using this heuristic

method is it will reduce the number of recursive calls required till reach an invalid state. This means

when unwinding all the way back to the first recursive call for the first position tested, it will do so in

less steps, so the algorithm will be able to calculate when the grid does not have a solution faster

than just choosing to test the next position in order.

I chose to represent the separate sudoku states as objects. Each of the objects have attributes to store their own:

* current grid values - copied from the numpy array passed in
* possible values -A 3d array that would be indexed for every position in the grid. At each position it would hold an array of all the possible values for that cell. If the cell had already been correctly assigned then it would have an empty array for its possible values.
* Unknown positions – this was used to store the x,y positions in the grid of all the values that hadn’t yet been assigned i.e. = 0

I decide to implement this as a class so that I could execute multiple methods upon the current

attributes of that state. Having the separate objects also helped to encapsulate the relevant date for

each state and to make sure that it did not affect the other states. The disadvantage of using objects

is it can reduce efficiency because, the depth first search algorithm requires lots of the new objects

to be instantiated. I tried to reduce the toll of the instantiations by passing the ‘possible values’ and

‘unknown values’ from the previous grid to the new one so that they wouldn’t have to be

recalculated each time. (Previously, I was recalculating the possible values each time an object was

instantiated but when I realised I could pass the ‘possible values’ and ‘unknown values’ and then

make a deep copy of them it in \_\_innit\_\_ it saved so much time!) Just had to remember to make a

copy of the array in \_\_innit\_\_ because even though the arrays are passed in by value, they are

assigned by reference. So without the copy then would be altering the possible values of a previous

state and the values wouldn’t get properly encapsulated.

The class methods I implemented to be carried out on the sudoku states included:

* Calc possible values – calculates the possible values for a specific cell. This is calculated by passing through the row, column and square of the cell and removing any numbers that it finds in the pass from the array of possible values for that cell. Only leaving the possible values that the cell can take.
* Calc All possible values – uses the calc possible method but instead of just applying it to a singular cell it calculates the possible values for the entire grid
* Set value – sets a given cell position to a given value in the grid and updates the attributes of the sudoku state to reflect the changes.
* remove possible – (called inside set value) updates the possible value of all of the unknown cells that may have been affected by the new value being set. (Basically tidies up after the value has been set)
* Fix singletons – finds any unknown values with only 1 possible value and automatically sets their value. With the assumption that if there is only one remaining possible value then it must be the correct value for that unknown cell. This helps make the algorithm more efficient because it means that if the algorithm is on the right track to find a goal state then it will do so faster, because it can assign more values per recursive call. It also works to find out if a grid doesn’t have a solution faster because it is more likely to reach an invalid state faster for the same reason.
* Set value 2 – if the method that is actually called in the depthfirst search and is used to combine both the set value and fix singletons so that set value doesn’t become too recursive.
* Is invalid – test if the state of the grid is invalid by checking if a cell that is still unknown has no possible values remaining to try. i.e. there is state that cant have a value
* Grid contains – just passes through the whole grid to check if it contains a given value
* Correct grid – loops through every row in the grid and checks that every number 1-9 is in every row. (an additional check for the gaol condition)
* Is goal – checks that the list of unknown positions is empty and that there are no more 0s left in the grid. Then as final check, makes sure that the grid is ‘correct’ (see above)
* Getters – I just made some getters for some of the attributes needed in the backtracking algorithm