Read me

A text file that explains your approach and the decisions you made in your own words – a readme file

* Submissions that do not include the written section will receive zero marks – **this part is mandatory**
* You may write your file in plain text (.txt) or [Markdown](https://www.markdownguide.org/basic-syntax/) (.md)
* To get top marks on this assignment, as well as getting a high grade from your implementation, you must also demonstrate excellent academic presentation in your written section

Summary of the algorithm and the logic on how it works:

For this Sudoku solver coursework, I have implemented a backtracking depth-first search using a

sudoku objects to represent its state. The backtracking depth-first search algorithm is based on the

algorithm from the course notes except I added an additional method that will choose the next

unknown position to test based on which position in the grid has the least ‘possible values’ (where ‘possible values’ means: the numbers out of 1-9 that the cell can be set to based on the numbers that in that cell’s row, column and 3x3 grid contain).

Reason for choosing the next position to be the one with the least ‘possible values’ is that it will

reduce the number of value iterations that the search will have to complete before it hits an invalid

state. This means that it will also reduce the number of recursive calls required. This means when

unrolling all the way back to the call for the first position tested, it will do so in less steps, so that you

know when the grid does not have a solution faster than just choosing to test the next position in order.

Each of the sudoku states that are used in the depth first search are represented by objects

instantiated by the class ‘Sudoku\_State’. 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. There is a slight disadvantage to this efficiency wise 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!)

Some of the class’s methods include: ‘calc\_possibleVals’, ‘set\_value’, ‘is\_invalid’, ‘is\_goal’ which are

pretty self-explanatory. Other less obvious methods include:

* ‘Calc\_AllPossibles’ this method calculates the ‘possible values’ for all the cells on the grid,

rather than just calculating the ‘possible values’ that will need updating because of the

position that was just set. This method is called when no ‘possible values’ have been created

for that grid yet or if the ‘possible values’ need to be recalculated.

* ‘remove\_possibles’ is called within ‘set\_value’ and is essentially the tidying up algorithm. It

goes through all the ‘possible values’ of the cells in the same row, column and square that

has just been changed and removes the value that the cell was set to from all of their possible values.

This makes the algorithm more efficient because it only tidies the cells that could have been

affected by the newest value set rather than all of the ‘possible values’

* ‘Fix\_Singletons’ is a method that is used to automatically assign any values on the grid that

only have one possible value remaining. The purpose of this is it will reach the goal state

faster because you can be sure that if there is only one possible value remaining for a cell

then it is either correct or will quickly cause an invalid state that can be used to backtrack.