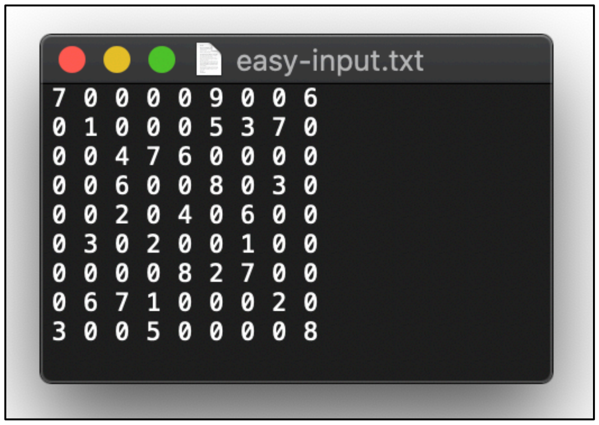
# COMS516X Homework 1

## Part I: Program

The program is a sudoku solver that is created using Hill Climbing algorithm. The program is written in python and will take in a real Sudoku problem as .txt file while 0's is used to represent blank spaces and numbers from 1-9 as the problem. Below is an example of the content of a valid input file:



easy-input.txt

```
7 0 0 0 0 9 0 0 6
0 1 0 0 0 5 3 7 0
0 0 4 7 6 0 0 0 0
0 0 6 0 0 8 0 3 0
0 0 2 0 4 0 6 0 0
0 3 0 2 0 0 1 0 0
0 0 0 0 8 2 7 0 0
0 6 7 1 0 0 0 2 0
3 0 0 5 0 0 0 0 8
```

*Figure 1 Example of a valid input file to the program.*

The sudoku solver also take the "bad move" probability as an input, where bad move probability is used here to allow probability for making a bad move. Its value is set to 0.001 if not defined by user. We will see in the experiment results of Part II to why it is important to allow some probability of taking a bad move in Hill Climbing algorithm.

The program will then run iteratively to look for the solution and output both final solution and number of iterations taken.

**<u>How to run</u>**

To run the program with command prompt, go to the directory with both the sudokuSolver.py and an input file. Make sure to have the python packages (numpy, matplotlib) needed for the program to run. Run the following command, where "`easy-input.txt`" is the input file name and `0.01` is the bad move probability (if this argument is missing program will default the value to 0.001).

```
python sudokuSolver.py easy-input.txt 0.01
```

*All results are from runs using "easy-input.txt" as input file.

## Part II: Experiments

### Algorithm Design

Transformation Function

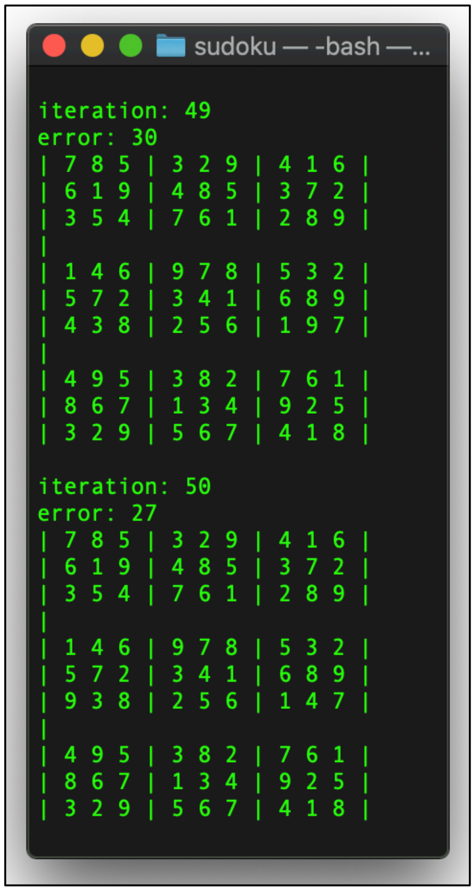
The transformation function used by this algorithm is simple, each iteration, **a row is chosen randomly to have 2 of its cells swapped**, choice of cells is also random, excluding the cells that are predefined from the input.

Fitness Function

The fitness function used is determined by the error count in the Latin squares. The goal is to **minimize the error count to 0**. It looks at each column and the sub 3x3 Latin square. Here we are not considering any error within a row as the Latin squares are randomly initialized in a way such that each row will have all 1-9, and the transformation function only swap 2 cells within a row. **Every missing number in each column or sub-square counts as 1 error, and every overlapping number with the predefined number in each column or sub-square counts as 3 errors**. The error count is tripled in the latter case to prioritize not overlapping with the predefined number.

Stopping Criterion

The algorithm will end when the fitness function achieved its goal, or in other words, the **error count is 0**, or when it has **exceeded 750000 tries in searching for a better state** to move to, or when it has **exceeded 75000 iterations of moving to the new state**. The limit on the number of tries and iterations is to make sure that the program does not run forever or take too long to run.



```
iteration: 49
error: 30
| 7 8 5 | 3 2 9 | 4 1 6 |
| 6 1 9 | 4 8 5 | 3 7 2 |
| 3 5 4 | 7 6 1 | 2 8 9 |
|
| 1 4 6 | 9 7 8 | 5 3 2 |
| 5 7 2 | 3 4 1 | 6 8 9 |
| 4 3 8 | 2 5 6 | 1 9 7 |
|
| 4 9 5 | 3 8 2 | 7 6 1 |
| 8 6 7 | 1 3 4 | 9 2 5 |
| 3 2 9 | 5 6 7 | 4 1 8 |

iteration: 50
error: 27
| 7 8 5 | 3 2 9 | 4 1 6 |
| 6 1 9 | 4 8 5 | 3 7 2 |
| 3 5 4 | 7 6 1 | 2 8 9 |
|
| 1 4 6 | 9 7 8 | 5 3 2 |
| 5 7 2 | 3 4 1 | 6 8 9 |
| 9 3 8 | 2 5 6 | 1 4 7 |
|
| 4 9 5 | 3 8 2 | 7 6 1 |
| 8 6 7 | 1 3 4 | 9 2 5 |
| 3 2 9 | 5 6 7 | 4 1 8 |
```

*Figure 2 Snapshot of a single iteration*

**Example** of a single iteration

Figure 2 shows the snapshot of a single iteration or transformation where the new state has a lower error count. This is the real sudoku problem in “easy-input.txt”. The transformation happened on the sixth row, where number 4 and 9 are swapped. This transformation resulted in 3 less error count as number 4 causes 2 errors since there is another 4 in the same sub-square and the same column, and number 9 causes 1 error as there is another 9 in its sub-square. Swapping 4 and 9 in this row elevate both problems, thus resulting in 3 less errors. More detailed on how to count the errors can be seen in the Appendix.

*All results are from runs using “easy-input.txt” as input file.

## <u>Experiment Results</u>

### <u>Move only when the solution is better</u>

We have tested two slightly different variation of the algorithm, approach (1): only move to a new state if the new state is better than the current state, approach (2): moves to the new state when the new state is better or has the same fitness as the current state. With "easy-input.txt" and bad move probability of 0.001 as the input, Table 1 shows the average number of iterations taken to reach the final solution for both approaches. 0.001 bad move probability is used here to better showcase the differences between both approaches, as not allowing bad moves at all often cause the program to reach an infinite loop as explained in later section.

*Table 1 Number of iterations taken to reach the final solution with different variation of Hill Climbing algorithm.*

| Approach\Run | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Ave |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. better | 8560 | 125 | 2911 | 3807 | 7054 | 10668 | 9564 | 20415 | 29672 | 13010 | 10579 |
| 2. same or better | 13912 | 4305 | 865 | 20875 | 3568 | 14030 | 4663 | 4606 | 5668 | 2087 | 7458 |

As we can see in Table 1, **approach 2 is better than approach 1** by requiring 30% less iterations to reach the solution. This might be because of situation where swapping 2 cells might result in the same error count, but the new state allows a better next move while the current state might be the local minimum. When such situation occurs, approach 1 prohibits moving to the better state with the same number of errors and making a bad move might mean moving to a new state with worse error count, thus requiring more iterations to reach the final solution.

### <u>Probability for making a bad move</u>

We have also tested the program with different bad move probabilities and observed the amount of time and number of iterations takes for it to converge to the final solution. Table 2 shows the average number of iterations (based on 10 runs, and iteration here only considers iteration where the state is updated) taken for the algorithm to reach a final solution, range of number of iterations, probability for the algorithm to reach the actual solution, and average time taken for the program to terminate, given a different value of bad move probability.

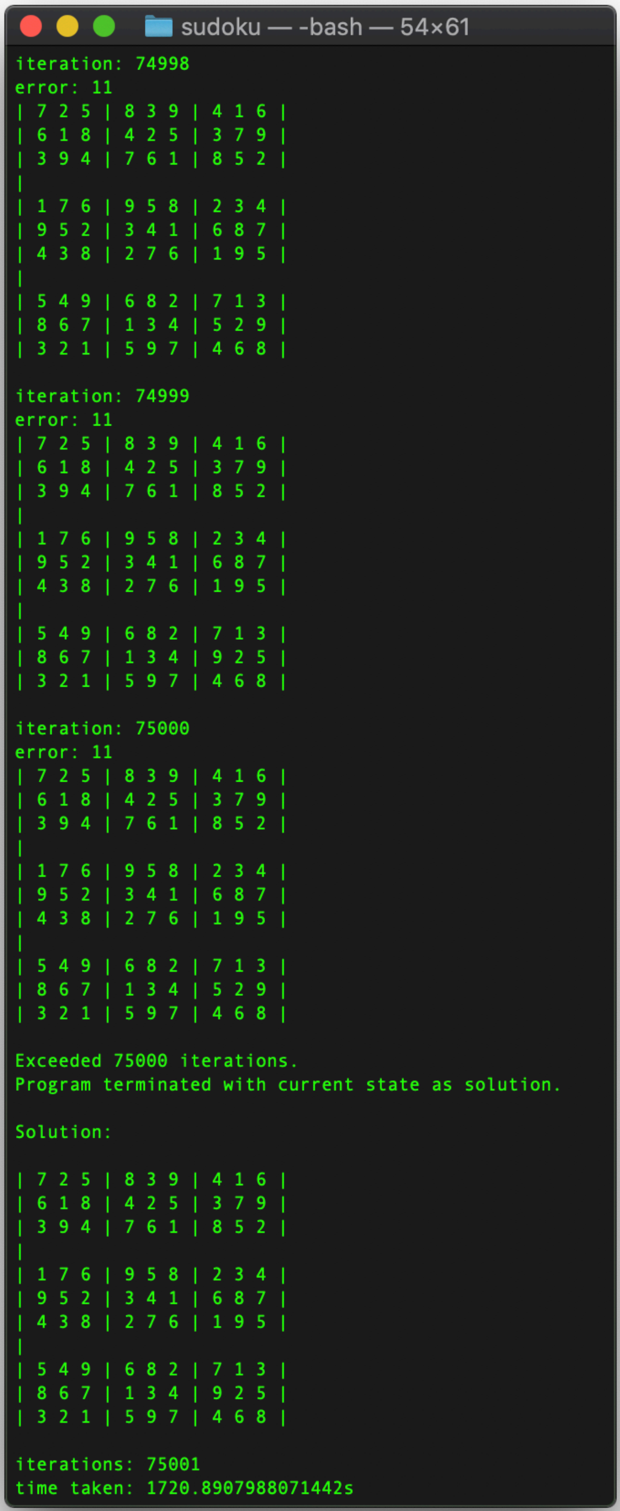
*Table 2 Average number of iterations taken to reach the final solution with different bad move probability. Values are represented as "a/b" where a is the result with aprroach 1 and b is the result with approach 2.*

| Bad move probability | 0 | 0.0001 | 0.001 | 0.01 | 0.1 |
|---|---|---|---|---|---|
| Average number of iterations | 37/72 | 11562/28143 | 10579/7458 | 15622/11062 | - |
| Range of iterations | [33,44]/[65,83] | [1373,39410]/[1911,55679] | [125,29672]/[865,20875] | [3152,42443]/[845,20663] | - |
| Reach the actual solution | 0% | 100% | 100% | 90%/100% | 0% |
| Average time taken (s) | - | 3498/637 | 1464/145 | 1250/196 | - |

*All results are from runs using "easy-input.txt" as input file.

As we can see in Table 2, it is **important to at least allow some “bad moves”** to be taken as Hill Climbing algorithm does not guarantee finding the global maximum (or minimum in this case) as the final solution. Figure 3 shows an example of a local minimum where there are obvious errors in the sudoku but swapping any 2 numbers within any row will only result in a higher error count. Another interesting finding is that with approach 2, by not allowing any bad move, it may occur a situation where the algorithm is stuck in a local minimum, constantly swapping the same 2 cells back and forth, and gets terminated when it reached the maximum number of iterations. This case is also shown in Figure 3, where number 5 and 9 at row 8 are constantly being swapped once it converged to 11 errors.

The program is unable to reach the actual solution when the bad move probability is 0.1 as it always exceeds the maximum iterations allowed. Here we can conclude that **allowing too many mistakes makes it hard for the program to quickly reach the best solution**. With the “easy-input.txt” as input, we can see that both 0.01 and 0.001 takes about the same number of iterations to reach the actual solution. However, it does take the program slightly more iterations to reach the final solution with bad move probability of 0.01, which further support the previous claim.



```
iteration: 74998
error: 11
| 7 2 5 | 8 3 9 | 4 1 6 |
| 6 1 8 | 4 2 5 | 3 7 9 |
| 3 9 4 | 7 6 1 | 8 5 2 |
|
| 1 7 6 | 9 5 8 | 2 3 4 |
| 9 5 2 | 3 4 1 | 6 8 7 |
| 4 3 8 | 2 7 6 | 1 9 5 |
|
| 5 4 9 | 6 8 2 | 7 1 3 |
| 8 6 7 | 1 3 4 | 5 2 9 |
| 3 2 1 | 5 9 7 | 4 6 8 |

iteration: 74999
error: 11
| 7 2 5 | 8 3 9 | 4 1 6 |
| 6 1 8 | 4 2 5 | 3 7 9 |
| 3 9 4 | 7 6 1 | 8 5 2 |
|
| 1 7 6 | 9 5 8 | 2 3 4 |
| 9 5 2 | 3 4 1 | 6 8 7 |
| 4 3 8 | 2 7 6 | 1 9 5 |
|
| 5 4 9 | 6 8 2 | 7 1 3 |
| 8 6 7 | 1 3 4 | 9 2 5 |
| 3 2 1 | 5 9 7 | 4 6 8 |

iteration: 75000
error: 11
| 7 2 5 | 8 3 9 | 4 1 6 |
| 6 1 8 | 4 2 5 | 3 7 9 |
| 3 9 4 | 7 6 1 | 8 5 2 |
|
| 1 7 6 | 9 5 8 | 2 3 4 |
| 9 5 2 | 3 4 1 | 6 8 7 |
| 4 3 8 | 2 7 6 | 1 9 5 |
|
| 5 4 9 | 6 8 2 | 7 1 3 |
| 8 6 7 | 1 3 4 | 5 2 9 |
| 3 2 1 | 5 9 7 | 4 6 8 |

Exceeded 75000 iterations.
Program terminated with current state as solution.

Solution:

| 7 2 5 | 8 3 9 | 4 1 6 |
| 6 1 8 | 4 2 5 | 3 7 9 |
| 3 9 4 | 7 6 1 | 8 5 2 |
|
| 1 7 6 | 9 5 8 | 2 3 4 |
| 9 5 2 | 3 4 1 | 6 8 7 |
| 4 3 8 | 2 7 6 | 1 9 5 |
|
| 5 4 9 | 6 8 2 | 7 1 3 |
| 8 6 7 | 1 3 4 | 9 2 5 |
| 3 2 1 | 5 9 7 | 4 6 8 |

iterations: 75001
time taken: 1720.8907988071442s
```

*Figure 3 Algorithm stuck in a local minimum. Program terminated after exceeding maximum number of iterations as defined.*

<u>Other experiments/findings</u>

Transformation function
A slightly different transformation function was used, which is to swap 4 numbers instead of just 2, and it is quickly found that this might cause the algorithm in failing to reach the final solution. For example, in the situation

*All results are from runs using “easy-input.txt” as input file.

where only 2 numbers are needed to be swapped to reach the final solution but since this transformation function always require 4 numbers to be swapped in each iteration, the program will either get stuck in this state indefinitely while trying to find for a better state, or it had to make a bad move and stray further away from the final solution.
The starting point of the Hill Climbing process also affects how long it takes for the program to reach the endpoint. With that in mind, another simple modification to the program can also be made by adding another process to the beginning of the program, such that before starting the Hill Climbing process, the program should produce some amount of randomly filled sudokus from the input, then proceed with the one with least error as its starting point.

Fitness function
Initially, the fitness function used was only to check how many missing numbers there are in each column and sub-square. It is then found out that this fitness function is too coarse as none of the runs converged. To refine the fitness function, error count for the overlapping number with predefined numbers is then added to consideration of total error count. After some trial and error, it is decided that a number that overlaps with a predefined number has its error count tripled of what a missing number would contribute. This adjustment later showed to help the algorithm to converge faster as it placed more priority on moving to a more probable new state.

Bad move probability
The bad move probability is a hyperparameter that should be set depending on how hard the sudoku problem is. An easier problem should have a lower bad move probability to ensure that the program can converge quickly, as an easier problem would have more predefined numbers and less blanks to be filled it, thus less possible error. We hypothesize the following: with an easier problem, the program would work better trying to converge most of the time; while a harder problem will most likely favor from having a higher bad move probability, as harder problem would have less predefined numbers and more blanks to be filled, thus more datapoint for a local minimum. Intuitively, allowing the program to make a decent amount of bad moves would shorten the search time as this would allow the algorithm to move away from a local minimum quicker.

Stopping criterion
The maximum number of iterations is also a hyperparameter that should be set depending on the bad move probability. The higher the bad move probability, the higher the stopping criterion should be set to, to allow for more tries as bad moves are made more often. For example, in Table 2, we can see that with bad move probability as 0.01, there exists 1 out of 10 runs that failed to reach the actual solution. With unlimited time, or unbounded number of iterations allowed, any Hill Climbing algorithm with some bad move probability will eventually reach the actual solution. (To change the maximum number of iterations or number of tries in searching for a good state allowed, user would have to open the python script and update the variables named "`iter_max`" or "`tries_max`" in the main function)

*All results are from runs using "easy-input.txt" as input file.

## Conclusion

With this Sudoku Solver, we can see that Hill Climbing algorithm does not guarantee to reach the optimum solution, as shown with the cases where the bad move probability is 0. Higher the number of local minimums, the lower the probability for Hill Climbing to reach the actual solution. From the experiment results, we can conclude that this Sudoku Solver works the best on easy sudoku problems by considering moving to a state with the same amount of error count as a good move, and with a bad move probability of 0.001. This Sudoku Solver can be improved in many ways, such as using a better transformation function, or a smarter search algorithm such as Simulated Annealing.

*All results are from runs using “easy-input.txt” as input file.

## Appendix

Counting the error
If we look at iteration 50 in Figure 2, we can count the errors as follow:

Error count from each column

- column 1: 1 missing number, 1 number overlapped with predefined number = 1+3(1) = 4
- column 2: no error = 0
- column 3: 2 missing numbers = 2
- column 4: 2 missing numbers = 2
- column 5: 2 missing numbers + 2 numbers overlapped with predefined numbers = 2+2(3) = 8
- column 6: 1 missing number = 1
- column 7: 1 missing number = 1
- column 8: 2 missing numbers = 2
- column 9: 2 missing numbers = 2
- total: 4+0+2+2+8+1+1+2+2 = 22

Error count from each sub-square

- square 1: 1 missing number = 1
- square 2: no error = 0
- square 3: 1 missing number = 1
- square 4: no error = 0
- square 5: no error = 0
- square 6: no error = 0
- square 7: 1 missing number = 1
- square 8: 1 missing number = 1
- square 9: 1 missing number = 1
- total: 1+0+1+0+0+0+1+1+1 = 5

Hence the total number of errors is 27.

*All results are from runs using “easy-input.txt” as input file.