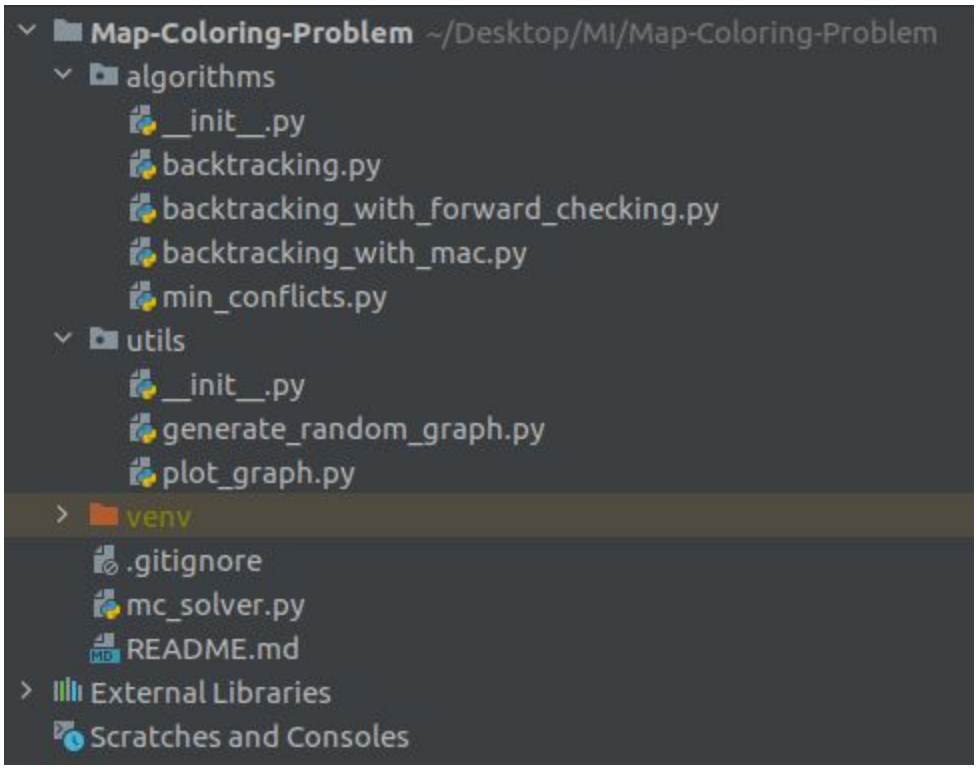
Map Coloring(Problem 6) Report

Amr Ahmed Abdelbaqi Mahmoud
Section:2, Bn:3

Code Structure:

The code contains 3 main modules:

- mc_solver.py which refers to map coloring solver.
- Algorithms folder which contains 4 algorithms used to solve this problem (backtracking, backtracking with forward checking, backtracking with maintaining arc consistency, min-conflicts)
- Utils folder which contains 2 utilities used to help solving this problem (generate random graph(which in our case generate problem), plot graph to visualize the problem)



Description:

- mc_solver.py is the interface which we run to solve the problem we give to it the parameters that we want our problem to be solved with.
- We have 5 parameters => number of nodes in the graph, number of colors used, algorithm used to solve the problem, max steps used by min conflicts algorithm, number of runs for the same previous parameters(This used to when generating statistics)

- mc_solver.py then calls generate random graph to generate the problem then use the specified algorithm in the parameters to solve the problem.
- mc_solver.py plot the graph before solving and after finding a solution, sometimes the problem may not have a solution.
- Each algorithm file contains the functions used to solve the problem using this algorithm.
- Each algorithm returns the color assignment of each node in the graph if a solution exists.

How to run and required dependencies:

To run the algorithm you will need to install the following dependencies

- click
- networkx
- matplotlib
- And you may need to install this sudo apt-get install python3-tk
  If this warning appeared “UserWarning: matplotlib is currently using agg, which is a non-gui backend. So cannot show the figure”

Firstly, you can run this command to know how to initialize the parameters
`python mc_solver.py --help`

```
(venv) (base) amr@Amr-Laptop:~/Desktop/MI/Map-Coloring-Problem$ python mc_solver.py --help
Usage: mc_solver.py [OPTIONS]

  Map coloring problem solver

Options:
  -a, --algorithm [mc|bt|bt-fc|bt-mac]
                                  Algorithm type.  [required]
  -k, --k_coloring INTEGER        Number of colors to solve this problem(only
                                  3 and 4).  [required]

  -ms, --max_steps INTEGER        Number of maximum steps used by min
                                  conflicts algorithm.  [required]

  -nr, --number_of_runs INTEGER   Number of runs with the same parameters.
                                  [required]

  -n, --number_of_nodes INTEGER   Number of nodes used in the map coloring
                                  [required]

  --help                          Show this message and exit.
```

Here is an example of a command with there parameters
(algorithm = backtracking, number of nodes = 10 , k = 4, max steps and number of runs are left to default)

```
python mc_solver.py -a bt -k 4 -n 10
```

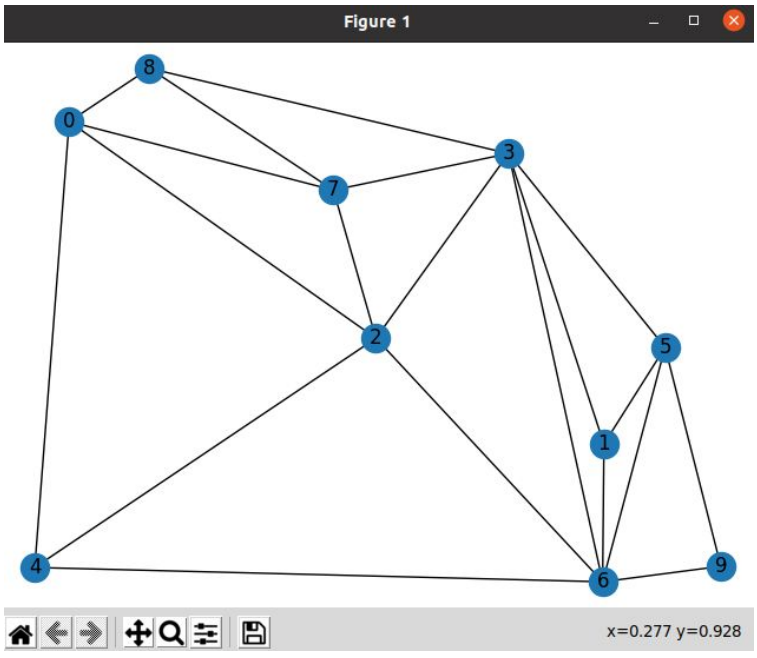
Expected output:

The output should be the graph before coloring, and the graph after coloring if there is an answer exists.
Note: the graph before coloring should be closed to see the graph after coloring.

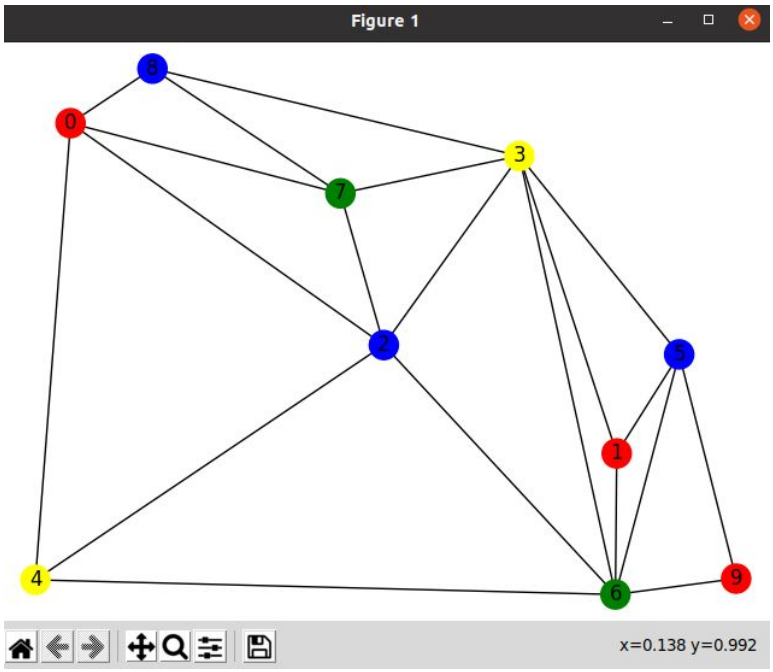
Or the output may be “No Solution exists.” in the terminal if no solution exists.
Note: in case of min conflicts algorithm sometimes no solution exists due to max step constrain

Example 1 output:

```
python mc_solver.py -a bt -k 4 -n 10
```



Graph before solving the problem



Graph after solving the problem

```
(venv) (base) amr@Amr-Laptop:~/Desktop/MI/Map-Coloring-Problem$ python mc_solver.py -a bt -k 4 -n 10
Solving for these parameters:  Algorithm= bt , Number of colors= 4 , Number of nodes= 10
Number of runs= 1
Color Assignment [0, 0, 1, 2, 2, 1, 3, 3, 1, 0]
Run Time =  2.0329000108176842e-05
```

Terminal

Example 2 output:

Used for the results section.
If number of runs parameter is given then the expected output is

```
(venv) (base) amr@Amr-Laptop:~/Desktop/MI/Map-Coloring-Problem$ python mc_solver.py -a bt-fc -k 4 -n 10 -nr 10
Solving for these parameters:  Algorithm= bt-fc , Number of colors= 4 , Number of nodes= 10
Number of runs= 10
Average Run Time =  1.067920006789791e-05 , Percentage of finding a solution =  100.0 %
(venv) (base) amr@Amr-Laptop:~/Desktop/MI/Map-Coloring-Problem$
```

Briefly explain how you wrote the code and how it helps you solve the problem section:

I started implementing the algorithms first and tested them using a fixed graph, firstly I started with backtracking and min conflicts then added forward checking and then mac. This approach helped me to focus on solving the problem as I needn't to think about generating the problem first. Here is the github repo link which will illustrate more the sequence I used to implement the code

https://github.com/3amrA7med/Map-Coloring-Problem

Results and Conclusions:

All results are obtained by running the same algorithm 10 times with random graphs to the same number of nodes.
The first number is the average run time and the second number is the percentage of finding a solution.

Min conflicts (max step = 100)

Number of nodes\k	K = 3 Avg time / solution found%	K = 4 Avg time / solution found%
N = 5	0.000121 / 60%	1.6534e-05 / 100 %
N = 10	0.00037 / 10%	9.62985e-05 / 90%
N = 30	0.00081 / 0%	0.000928 / 20%
N = 50	0.0012 / 0%	0.00165 / 0%
N = 70	0.00163 / 0%	0.00211 / 0%

For min-conflicts algorithm its totally random I have tried to solve it different times and answers are with wide variety sometimes repeated sometimes not if we repeated the test for the min-conflicts the avg runtime won't differ much but the percentage of solving will vary

Backtracking

Number of nodes\k	K = 3 Avg time / solution found%	K = 4 Avg time / solution found%
N = 5	4.48369e-06 / 100%	2.87031e-05 / 100%
N = 10	0.000143 / 10%	1.59026e-05 / 100%
N = 30	0.0466 / 0%	0.006841 / 100%
N = 50	6.6522 / 0%	169.391 / 100 % For one run only not 10
N = 70	Not calculated but expecting that there will be no answers and the avg time will be much larger than precious step	Not calculated but expecting to be solved in much more time

For backtracking algorithm it is efficient for small values of N and the average run time is acceptable and its efficient as the number of recursion needed is small so it should like forward checking and mac algorithms or sometimes it's better as they have an overhead added for each function call (for the same graph)so sometimes the overhead of the recursion is less than the overhead in the forward checking and mac.
But for higher values of N, backtracking becoming in-efficient due to high number of recursions needed to solve the problem, as we see in the table for N=50 or higher the algorithm tends to need a lot of time so in this case the overhead of the forward checking and mac will be much less than recursion calls number needed by backtracking

Backtracking with forward checking

Number of nodes\k	K = 3 Avg time / solution found%	K = 4 Avg time / solution found%
N = 5	8.7077e-06 / 80 %	5.64900e-06 / 100%
N = 10	5.4969e-05 / 10%	1.36657e-05 / 100%
N = 30	0.00197 / 0%	0.00570 / 100 %
N = 50	0.016649 / 0%	1.53468 / 100 %
N = 70	0.2780 / 0%	56.151 / 100 % for one run not 10

For Backtracking with forward checking algorithm it outperform the normal backtracking for higher values of N but when it reached 70 it struggles for the number of recursion needed so the runtime is almost one minute for only one run not 10.

Backtracking with mac

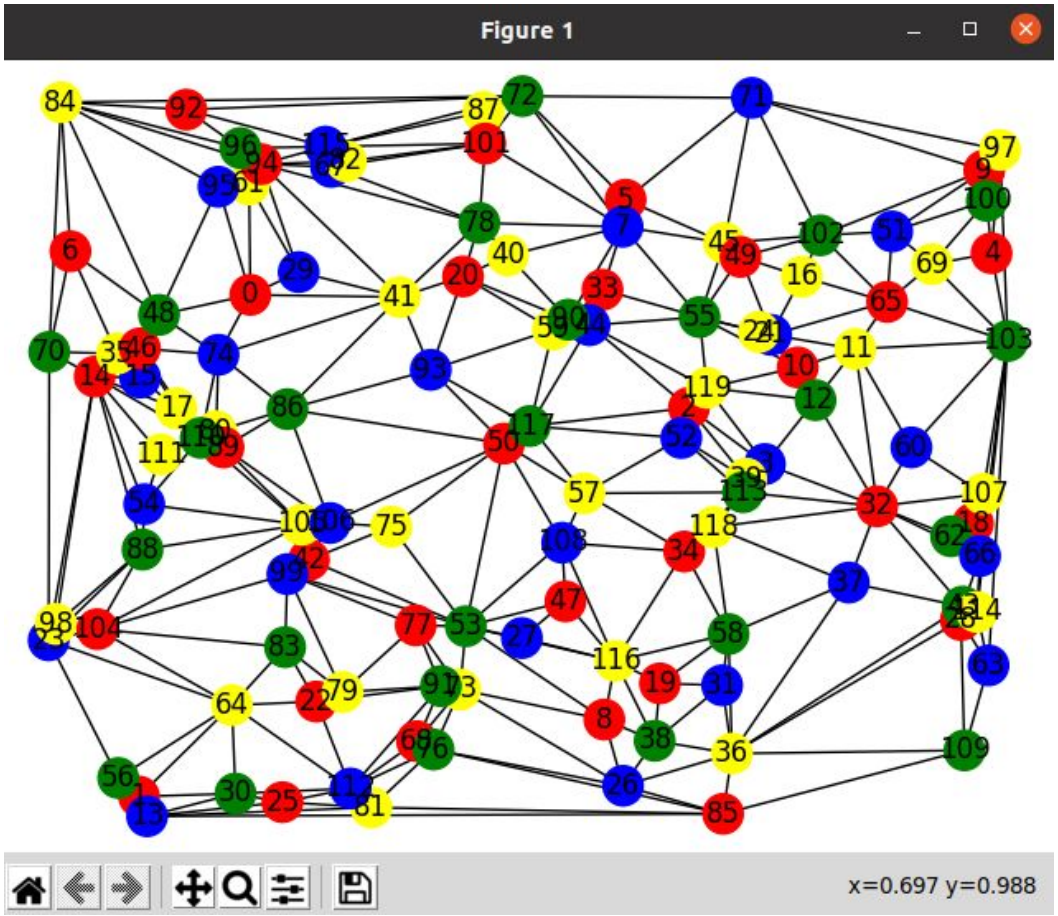
Number of nodes\k	K = 3 Avg time / solution found%	K = 4 Avg time / solution found%
N = 5	6.43767e-05 / 70 %	3.7934e-05 / 100%
N = 10	0.000198 / 20%	0.0001958 / 100%
N = 30	0.001433 / 0 %	0.00333 / 100%
N = 50	0.00489 / 0%	0.07044 / 100%
N = 70	0.0120831 / 0%	0.451294 / 100%

For backtracking with a mac it outperforms all the previous algorithms especially for high values of N as it solves N=70 with average runtime = 0.5 second ! that is a great improvement indeed

Note: I used the algorithm again for N=70 it solved 9 graphs pretty fast but one graph made it stuck so there are some cases in N=70 values that may need higher runtime but in general it solves the graphs in 0.5 second

Another note: I managed to solve problems up to 120,130 with mac algorithm but it depends on the graphs some runs produce very efficient solution and some runs may stuck, but N=70 is much more stable as for 10 runs it produces 0.5 second runtime

Example to N= 120




```
(venv) (base) amr@Amr-Laptop:~/Desktop/MI/Map-Coloring-Problem$ python mc_solver.py -a bt-mac -k 4 -n 120 -nr 1
Solving for these parameters:  Algorithm= bt-mac , Number of colors= 4 , Number of nodes= 120
Number of runs= 1
Color Assignment [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 2, 3, 1, 0, 1, 2, 2, 0, 0, 0, 1, 0, 1, 2, 0, 1, 1, 0, 1, 3, 1,
 0, 0, 0, 2, 2, 1, 3, 2, 2, 2, 0, 3, 1, 2, 0, 0, 3, 0, 0, 1, 1, 3, 1, 3, 3, 2, 3, 2, 1, 2, 3, 1, 2, 0, 1, 1, 0, 2
, 3, 1, 3, 2, 1, 2, 3, 0, 3, 2, 2, 2, 2, 3, 2, 0, 3, 2, 3, 0, 3, 3, 0, 1, 0, 1, 3, 2, 2, 1, 3, 0, 3, 3, 0, 2, 1,
2, 1, 3, 3, 2, 1, 3, 2, 1, 2, 3, 2, 2]
Run Time =  3.359818338999503
```

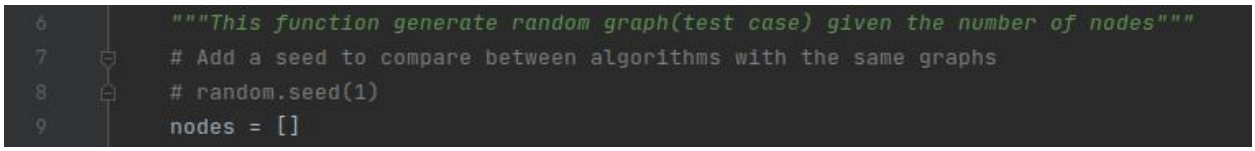
General conclusion:

For k = 3 (it depends on the random graph generated) but for N=10 only one or two from 10 runs have solution for all 4 algorithms, if we increase N to be more than 10 there will be no solutions
For k = 4 there will be a solution except if we are using min-conflicts as this is a random process and for higher values for N it will need higher max steps to investigate more random moves.
Backtracking with a mac layer is the most efficient in case of higher values of N but for small values it behaves the same as backtracking and forward checking and depending on the test case it can be less or more in the run time compared to these 2.

No assumptions have been made in the code, the problem has been implemented as asked in the document.

General Notes:

- If testing purpose is to generate statistics as in the results section we should comment line 8 from the generate random graph module(Because we need to generate random graphs not the same graph)



- If the testing purpose is to see the same specific graph solved by 4 different algorithms we should uncomment this line

Thank You For reading this