module-03-programming-only

February 9, 2019

1 Module 3 - Programming Assignment

1.1 Directions

There are general instructions on Blackboard and in the Syllabus for Programming Assignments. These are specific instructions for this assignment. Please read it carefully and make sure you understand what is being asked of you. Please ask questions on the discussion boards or email me at `EN605.645@gmail.com` if you do not understand something.

You must follow the directions *exactly* or you will get a 0 on the assignment.

You must submit your assignment as `<jhed_id>.py`.

The starter code is already set up in module03.py I have kept the directions as they were for the Notebook version so that you can see what is supposed to happen.

If there is a difference between this Notebook and `module03.py`, go with what is in `module03.py`.

```
In [3]: %matplotlib inline

        import matplotlib.pyplot as plt
        import networkx as nx
```

For this assignment only

If you want to use NetworkX with your assignment, you can do:

```
conda install networkx
```

or

```
pip install networkx
```

1.2 CSP: Map Coloring

In this programming assignment, you will be using your new understanding of **Constraint Satisfaction Problems** to color maps. As we know from the Four Color Theorem any division of a plane into contiguous regions can be colored such that no two adjacent regions are the same color by using only four colors.

From the book, we know that we can translate this problem into a CSP where the map is represented as a planar graph and the goal is to color all the nodes such that no adjacent nodes are colored the same color.

As with most AI problems, this requires us to figure out how best to represent the problem–and the solution–given the high and low level data structures and types at our disposal. For this problem, we'll settle on a Dict which contains at least two keys: "nodes" which is an *ordered* List of Strings that represents each node or vertex in the planar graph and "edges" which contains a List of Tuples that represent edges between nodes. The Tuples are of ints that represent the index of the node in the "nodes" list.

Using this system, and adding a "coordinates" key with abstract drawing coordinates of each node for NetworkX, we can represent the counties of Connecticut like so:

```
In [4]: connecticut = { "nodes": ["Fairfield", "Litchfield", "New Haven", "Hartford", "Middlese
                        "edges": [(0,1), (0,2), (1,2), (1,3), (2,3), (2,4), (3,4), (3,5), (3,6
                        "coordinates": [( 46, 52), ( 65,142), (104, 77), (123,142), (147, 85),
        print(connecticut)
```

```
{'nodes': ['Fairfield', 'Litchfield', 'New Haven', 'Hartford', 'Middlesex', 'Tolland', 'New Lo
```

The coordinates permit us to use NetworkX to draw the graph. We'll add a helper function for this, `draw_map`, which takes our planar_map, a figure size in abstract units, and a List of color assignments in the same order as the nodes in the planar_map. The underlying drawings are made by matplotlib using NetworkX on top of it. Incidentally, the positions just make the map "work out" on NetworkX/matplotlib.

The size parameter is actually inches wide by inches tall (8, 10) is an 8x10 sheet of paper. Why doesn't a chart cover up the whole screen then? It's adjusted by dpi. On high resolution monitors, 300 dpi with 8x10 inches might only take up a fraction of that space. Use whatever you want to make the output look good. It doesn't matter for anything else but that.

A default value for `color_assignments` is provided, `None`, that simply colors all the nodes red. Otherwise, `color_assignments` must be a `List of Tuples` where each `Tuple` is a node name and assigned color. The order of `color_assignments` must be the same as the order of `"nodes"` in the `planar_map`.

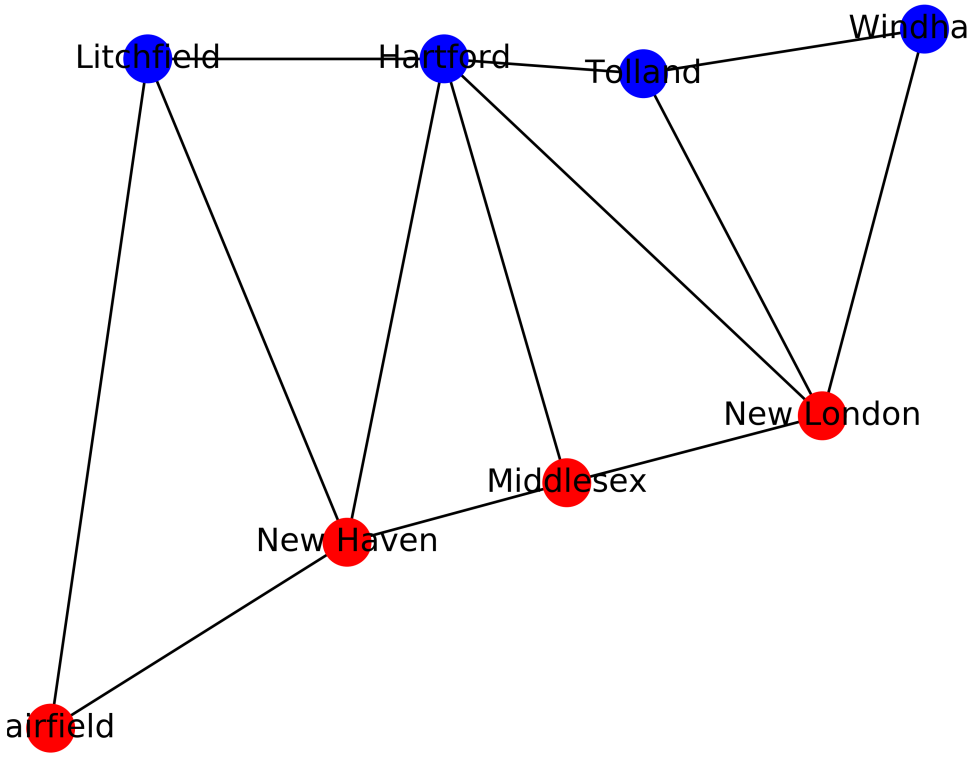
If you are using NetworkX in your assignment, you can copy this function into module03.py

```
In [5]: def draw_map(name, planar_map, size, color_assignments=None):
            def as_dictionary(a_list):
                dct = {}
                for i, e in enumerate(a_list):
                    dct[i] = e
                return dct

            G = nx.Graph()

            labels = as_dictionary(planar_map[ "nodes"])
            pos = as_dictionary(planar_map["coordinates"])

            # create a List of Nodes as indices to match the "edges" entry.
            nodes = [n for n in range(0, len(planar_map[ "nodes"]))]

            if color_assignments:
                colors = [c for n, c in color_assignments]
```

```
        else:
            colors = ['red' for c in range(0,len(planar_map[ "nodes"]))]

        G.add_nodes_from( nodes)
        G.add_edges_from( planar_map[ "edges"])

        plt.figure( figsize=size, dpi=600)

        nx.draw( G, node_color = colors, with_labels = True, labels = labels, pos = pos)

        plt.savefig(name + ".png")
```

Using this function, we can draw `connecticut`:

```
In [6]: draw_map("connecticut", connecticut, (5,4), [("Fairfield", "red"), ("Litchfield", "blue
                                                    ("Middlesex", "red"), ("Tolland", "blue"), ("New London"

/Users/stephyn/anaconda3/envs/en685648/lib/python3.7/site-packages/networkx/drawing/nx_pylab.py
  if cb.is_numlike(alpha):
```



This coloring obviously has some problems! You're going to write a program to fix it.

So what we (and by "we", I mean "you") would like to do is write a program that can figure out how to color a planar map...ie, `connecticut` *and* `europe`, you will do it by implementing a function that uses the algorithms that were discussed in this module.

1.3 Which CSP Algorithms?

You will need to implement **backtracking** and **forward checking**. You will also need to implement **Minimum Remaining Values** or **Degree Heuristic** (tell me which one) and **Least Constraining Value**. Break ties in ascending order (least to most).

I suggest you implement backtracking then forward checking then either minimum remaining values or degree heuristic then least constraining value. Try to submit a working program (if you don't make it to some requirement, comment out that code).

Please change the "?" below into "yes" or "no" indicating which elements you were able to implement:

```
backtracking: ?
forward checking: ?
minimum remaining values: ?
degree heuristic: ?
least contraining value: ?
```

Change the print calls in module03.py instead

Your function should take the following form:

```
def color_map( planar_map, colors, trace=False)
```

where `planar_map` has the format described above, colors is a List of Strings denoting the colors to use and `trace` operates as described in the next paragraph. It should return a List of Tuples where each Tuple is of the form `( Node Name, Color)` in the same order as the `node` entry in the planar_map. For example, if we had `["A", "B"]` as nodes and `["Yellow", "Green"]` as colors, your function might return `[("A", "Yellow"), ("B", "Green")]`. You can then use a List comprehension to extract the colors in order to feed them to `draw_map`. If a coloring cannot be found, return `None`.

Your function also will take an optional argument, `trace`, with a default value of `False`.

If `trace` is set to `True` your program will print out *traces* (or debugging) statements that show what it is currently doing (in terms of the algorithms you are supposed to implement). For example, if your program starts to backtrack, the trace should say that it has to backtrack and why.

As usual, you should implement your function using helper functions, using one Markdown cell for documentation and one Codecell for implementation (one function and assertions).

In module03.py, you can call your code like so:

```
python module03.py debug
```

and `True` will be passed to `trace` throughout the program.

```
In [ ]: def color_map( planar_map, colors, trace=False):
            return [(n, "red") for n in planar_map["nodes"]]
```

Currently, it just colors everything red. When you are done, if it cannot find a coloring, it should return `None`.

1.4 Problem 1. Color Connecticut Using Your Solution

This is already set up in module03.py for you

```
In [ ]: connecticut_colors = color_map( connecticut, ["red", "blue", "green", "yellow"], trace=
```

Using the "edges" list from the connecticut map, we can test to see if each pair of adjacent nodes is indeed colored differently:

```
In [ ]: edges = connecticut["edges"]
        nodes = connecticut[ "nodes"]
        colors = connecticut_colors
        COLOR = 1

        for start, end in edges:
            try:
                assert colors[ start][COLOR] != colors[ end][COLOR]
            except AssertionError:
                print "%s and %s are adjacent but have the same color." % (nodes[ start], nodes
```

```
In [ ]: draw_map( connecticut, (5,4), connecticut_colors)
```

```
In [ ]: connecticut_colors = color_map( connecticut, ["red", "blue", "green"], trace=True)
        if connecticut_colors:
            draw_map( connecticut, (5,4), connecticut_colors)
```

1.5 Problem 2. Color Europe Using Your solution

This is already set up in module03.py for you Run:

```
python module03.py debug
```

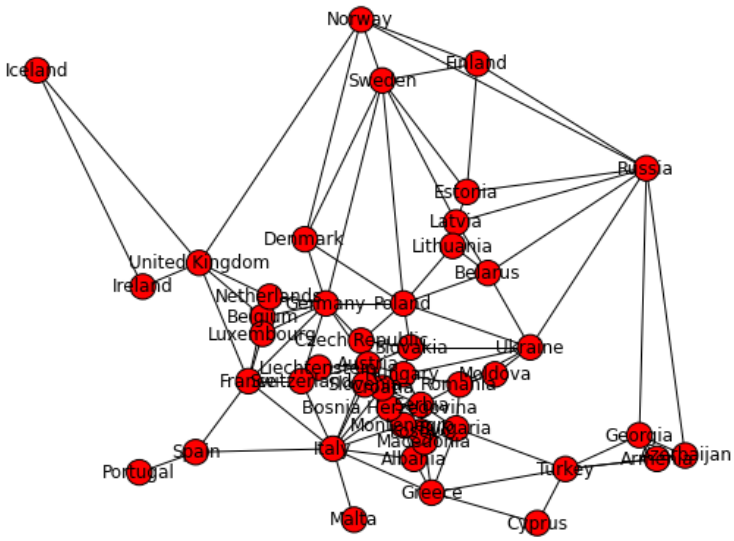
to get a sense of what will happen (or what you're going to keep from happening).

```
In [5]: europe = {
            "nodes":  ["Iceland", "Ireland", "United Kingdom", "Portugal", "Spain",
                       "France", "Belgium", "Netherlands", "Luxembourg", "Germany",
                       "Denmark", "Norway", "Sweden", "Finland", "Estonia",
                       "Latvia", "Lithuania", "Poland", "Czech Republic", "Austria",
                       "Liechtenstein", "Switzerland", "Italy", "Malta", "Greece",
                       "Albania", "Macedonia", "Kosovo", "Montenegro", "Bosnia Herzegovina",
                       "Serbia", "Croatia", "Slovenia", "Hungary", "Slovakia",
                       "Belarus", "Ukraine", "Moldova", "Romania", "Bulgaria",
                       "Cyprus", "Turkey", "Georgia", "Armenia", "Azerbaijan",
                       "Russia" ],
            "edges": [(0,1), (0,2), (1,2), (2,5), (2,6), (2,7), (2,11), (3,4),
                       (4,5), (4,22), (5,6), (5,8), (5,9), (5,21), (5,22),(6,7),
                       (6,8), (6,9), (7,9), (8,9), (9,10), (9,12), (9,17), (9,18),
                       (9,19), (9,21), (10,11), (10,12), (10,17), (11,12), (11,13), (11,45),
                       (12,13), (12,14), (12,15), (12,17), (13,14), (13,45), (14,15),
```

```
                        (14,45), (15,16), (15,35), (15,45), (16,17), (16,35), (17,18),
                        (17,34), (17,35), (17,36), (18,19), (18,34), (19,20), (19,21),
                        (19,22), (19,32), (19,33), (19,34), (20,21), (21,22), (22,23),
                        (22,24), (22,25), (22,28), (22,29), (22,31), (22,32), (24,25),
                        (24,26), (24,39), (24,40), (24,41), (25,26), (25,27), (25,28),
                        (26,27), (26,30), (26,39), (27,28), (27,30), (28,29), (28,30),
                        (29,30), (29,31), (30,31), (30,33), (30,38), (30,39), (31,32),
                        (31,33), (32,33), (33,34), (33,36), (33,38), (34,36), (35,36),
                        (35,45), (36,37), (36,38), (36,45), (37,38), (38,39), (39,41),
                        (40,41), (41,42), (41,43), (41,44), (42,43), (42,44), (42,45),
                        (43,44), (44,45)],
            "coordinates": [( 18,147), ( 48, 83), ( 64, 90), ( 47, 28), ( 63, 34),
                            ( 78, 55), ( 82, 74), ( 84, 80), ( 82, 69), (100, 78),
                            ( 94, 97), (110,162), (116,144), (143,149), (140,111),
                            (137,102), (136, 95), (122, 78), (110, 67), (112, 60),
                            ( 98, 59), ( 93, 55), (102, 35), (108, 14), (130, 22),
                            (125, 32), (128, 37), (127, 40), (122, 42), (118, 47),
                            (127, 48), (116, 53), (111, 54), (122, 57), (124, 65),
                            (146, 87), (158, 65), (148, 57), (138, 54), (137, 41),
                            (160, 13), (168, 29), (189, 39), (194, 32), (202, 33),
                            (191,118)]}
        print europe
```

```
{'nodes': ['Iceland', 'Ireland', 'United Kingdom', 'Portugal', 'Spain', 'France', 'Belgium', '
```

```
In [6]: draw_map( europe, (10, 8))
```




```
In [ ]: europe_colors = color_map( europe, ["red", "blue", "green", "yellow"], trace=True)
```

Here we're testing to see if the adjacent nodes are colored differently:

```
In [ ]: edges = europe["edges"]
        nodes = europe[ "nodes"]
        colors = europe_colors
        COLOR = 1

        for start, end in edges:
            try:
                assert colors[ start][COLOR] != colors[ end][COLOR]
            except AssertionError:
                print "%s and %s are adjacent but have the same color." % (nodes[ start], nodes
```

```
In [ ]: draw_map( europe, (10,8), europe_colors)
```

```
In [ ]: europe_colors = color_map( europe, ["red", "blue", "green"], trace=True)
        if europe_colors:
             draw_map( europe, (10,8), europe_colors)
```