CSE 30
Programming Abstractions: Python
Programming Assignment 6

In this project you will solve (approximately) the Graph Coloring (GC) problem discussed in class. Given a graph G, determine an assignment of colors from the set $\{1, 2, 3, \dots, k\}$ to the vertices of G so that no two adjacent vertices are assigned the same color. Further, try to reduce the size k of the color set as far as possible. Such an assignment is called a *k-coloring* of G, and G is said to be *k-colorable*. The smallest k for which G is k-colorable is called the *chromatic number* of G, and is denoted $\chi(G)$. Any graph with n vertices is necessarily n-colorable (just assign each vertex its own color). Therefore, any solution the GC problem is expected to satisfy $\chi(G) \le k \le n$. However, for a general graph G, the value $\chi(G)$ may not be known, so in solving GC, one is left not knowing what value of k is the minimum. Indeed, this is the difficulty.

Let us pause a moment and think about how we might solve this minimization problem exactly, i.e. find a k-coloring with $k = \chi(G)$. An assignment of colors from $\{1, 2, 3, \dots, k\}$ to $V(G) = \{1, 2, 3, \dots \dots, n\}$, is nothing but a mapping $f\colon \{1, 2, 3, \dots \dots, n\} \to \{1, 2, 3, \dots, k\}$. How many such mappings are there? This is a counting problem right out of Discrete Mathematics (CSE 16). There are k ways to choose the color $f(1)$, then k ways to choose the color $f(2)$, then k ways to choose $f(3)$, …, and finally k ways to choose $f(n)$. Altogether, the number of ways of making these choices in succession is:

$$\underbrace{k \cdot k \cdot k \cdot \dots \cdot k}_{n} = k^n$$

Therefore, the number of k-color assignments to $V(G)$ is k^n. Of course, not all of these mappings represent proper k-colorings, since adjacent vertices may be assigned the same color. Indeed, if $k < \chi(G)$, then no such mappings are k-colorings. Fortunately, it is easy to check whether a mapping is a k-coloring or not. Just step through all edges $\{x, y\} \in E(G)$, and check that the colors $f(x)$ and $f(y)$ are different. If for some $\{x, y\} \in E(G)$, we have $f(x) = f(y)$, then f is not a k-coloring.

A brute force solution to this minimization problem is now clear. For each $k = 1, 2, 3, \dots$, enumerate all k^n mappings $f\colon \{1, 2, 3, \dots \dots, n\} \to \{1, 2, 3, \dots, k\}$ (itself an interesting problem), then check each one to see if it is a proper k-coloring. The first (smallest) k for which this check succeeds is $\chi(G)$. Unfortunately, the algorithm just described is utterly impractical for all but the smallest n. Observe that the total number of checks performed is

$$\sum_{k=1}^{\chi(G)} k^n \ge \chi(G)^n,$$

which grows very rapidly with the size n of the vertex set of G. For instance, if G has $n = 100$ vertices and $\chi(G) = 10$, then more than 10^{100} checks must be performed. Even if each check could be performed in a billionth of a second (far too optimistic), the amount of time consumed would be

$$10^{91} \text{ seconds} = (3.168 \dots) \cdot 10^{83} \text{ years.}$$

For contrast, the current estimate of the age of the universe is only $(13.772) \cdot 10^9$ years. There are ways to improve the above algorithm, such as to consider only surjective mappings from {vertices} to {colors}, but nothing can overcome its basic inefficiency. If someone were to discover an efficient algorithm for the

Graph Coloring problem, it would be considered a major advancement in theoretical computer science, and would win its inventor one million dollars. See

https://en.wikipedia.org/wiki/P_versus_NP_problem

and

https://en.wikipedia.org/wiki/Graph_coloring#Computational_complexity

for details.

Obviously then, our goal in this project must be more modest. Instead, you will create a program that only *approximates* an optimal solution, and does it *efficiently*. There is a word for this: heuristic. A *heuristic* for an optimization problem is a computational process that returns an answer that is *likely to be* a good approximation to optimal, and is *likely to be* efficient in doing so. There may be a small set of problem instances for which the approximation is bad, and some for which the runtime is long. An *algorithm* on the other hand, is supposed to *solve all instances* of a problem, i.e. always return an optimal solution.

The heuristic you develop for the GC problem will be based on the reachable() and BFS() algorithms discussed in class. These algorithms are very efficient and can be run on large graphs with thousands or even millions of vertices. Both algorithms start at a source vertex and search outward from it, always processing the next vertex nearest the source. This makes sense for the SSSP problem, since the goal is to determine distances from the source. It may not make sense for the GC problem. Here is the general outline followed by both algorithms.

```
1.  start somewhere
2.  while some vertex has not been "processed" (whatever that may mean)
3.      pick the "best" such vertex $x$ (whatever "best" means)
4.      process $x$
5.      for each neighbor $y$ of $x$
6.          "update" the vertex $y$ (whatever "update" means)
```

Both reachable() and BFS() have a defined starting point, the source, which is one of the algorithm inputs. For the GC problem we can start anywhere. Is there a best place to start? That will be up to you. Each vertex x participates in a constraint with each of its neighbors y, namely that $\text{color}[x] \neq \text{color}[y]$. The number of constraints on $\text{color}[x]$ is the number of neighbors of x, also called the *degree* of x, denoted $\deg(x)$. Should we pick the largest degree (most constraints) to start? The smallest (least constraints)? Should we consider the degrees of the neighbors of x? Should we ignore all this and just pick a random starting vertex? These are all things for you to consider. Once you get a running program, you will be able to do experiments, altering your heuristic in various ways to see if you get better results.

What does it mean to "process" a vertex x? In this problem, it pretty clearly means assigning a color to x, although even this is open to interpretation. One thing your heuristic *must* do is to always produce a proper k-coloring of G. To this end you should, for each $x \in V(G)$, maintain a set $\text{ecs}[x]$ containing the *excluded color set* for x, i.e. the set of colors that have already been assigned to its neighbors, and therefore cannot be assigned to x. Since our goal is to use the smallest possible number of colors, the color we assign to x should always be the smallest color in the set $\{1, 2, 3, \dots, n\} - \text{ecs}[x]$, i.e. pick the smallest color that can be assigned. This assures that there will be no gaps in the set of colors used. In other words, if we manage to find a 5-coloring using the set $\{1, 2, 3, 4, 5\}$, but the color 3 is never assigned, then we could have achieved a 4-coloring.

If "process" x means to pick color$[x]$, then to "update" one of its neighbors $y \in$ adj$[x]$, should mean to add color$[x]$ to ecs$[y]$, so when it comes y's turn to be "processed", we know what colors *not* to assign.

Finally, what should "best" mean on line 3 of the outline? This decision is where you will have the most leeway to be creative in designing your heuristic. Here are some ideas. You may base the choice for what is "best" on degrees again, i.e. pick an unprocessed (uncolored) vertex of either highest or lowest degree. You might also pick one with the largest excluded color set, the idea being to put out the biggest fire first. Another approach would be to pick a vertex closest to your starting point. In this case, you should maintain a FIFO queue with the closest vertex at the front, just in BFS(). Then to "update" a neighbor y of x would entail adding it to the back of the queue, as well as updating ecs$[y]$. You can refine all of these strategies by combining them in various ways. For instance, pick x to be an uncolored vertex of maximum degree, then break ties by picking one with largest ecs$[x]$.

Program Specifications
You will write a module called **graph.py** containing a Graph class. This file should be based on the example **graphs.py** discussed at length in class (**note the different spellings**). Begin by removing anything not needed in this project, like the attributes `_distance`, `_predecessor` and `_component`, and functions like `findComponents()` and `getPredecessor()`. Add the following attributes to your Graph class.

`_color`: a dictionary whose keys are vertices `x`, and value `_color[x]`, the color of `x`
`_ecs`: a dictionary whose keys are vertices `x`, and value `_ecs[x]`, the excluded color set of `x`

You may add other attributes that you deem necessary for your heuristic strategy. Include functions that perform the actions described in their respective doc strings.

```
def Color(self):
   """
   Determine a proper coloring of a graph by assigning a color from the
   set {1, 2, 3, .., n} to each vertex, where n=|V(G)|, and no two adjacent
   vertices have the same color. Try to minimize the number of colors
   used. Return the subset {1, 2, .., k} of {1, 2, 3, .., n} consisting
   of those colors actually used.
   """
   pass
# end

def getColor(self, x):
   """ Return the color of x."""
   pass
# end
```

It is recommended (but not required) that you include a helper function as described below.

```
def _find_best(self, L):
   """Return the index of the best vertex in the list L."""
   pass
# end
```

The required function `Colors()` is of course the main event in this project. Again you may add other functions as you deem appropriate.

Write a client program called **GraphColoring.py** containing the following functions.

```
def CheckProperColoring(G):
   """
   Return True if no two adjacent vertices in G have like colors,
   False otherwise.
   """
   pass
# end

def main():

   # check command line arguments and open files

   # read each line of input file

   # get number of vertices on first line, create vertex list

   # create edge list from remaining lines

   # create graph G

   # Determine a proper coloring of G and print it to the output file

   """
   # Check that the coloring is correct
   print(file=outfile)
   msg = 'coloring is proper: {}'.format(CheckProperColoring(G))
   print(msg, file=outfile )
   """
# end
```

You may follow the outline in function main() if you like, though it is not required. The code after the final comment (triple quoted out) is for diagnostic purposes only, and intended for you to run your own tests. Do not include those commands in your submitted version.

A sample run of your program is given below.

```
$ python3 GraphColoring.py
Usage: $ python3 GraphColoring.py <input file> <output file>
$ python3 GraphColoring.py in
Usage: $ python3 GraphColoring.py <input file> <output file>
$ python3 GraphColoring.py in out
[Errno 2] No such file or directory: 'in'
Usage: $ python3 GraphColoring.py <input file> <output file>
$ python3 GraphColoring.py in1 out1
```

As you can see, your program should do basic checks of the command line inputs, as in previous projects. It will then read from an input file, and write to an output file, both named on the command line.

File Formats
An input file will contain a single positive integer on line 1, giving the number of vertices in a graph. Each subsequent line will contain two integers, separated by a space, giving the ends of an edge in the graph. A blank line in the file terminates the input, and no lines after it will be read. The following example represents a graph with 15 vertices.

```
15
1 7
2 3
2 6
3 7
4 10
5 9
5 10
6 7
6 11
7 12
8 13
9 13
9 14
10 15
14 15
```

One possible output file to match this input is given below.

```
3 colors used: {1, 2, 3}

vertex    color
----------------
 1         1
 2         1
 3         3
 4         1
 5         1
 6         3
 7         2
 8         2
 9         2
 10        2
 11        1
 12        1
 13        1
 14        1
 15        3
```

The first line of the output file gives the number of colors used, then lists the colors in set braces. This is followed by a blank line, then a table listing each vertex and its corresponding color. As usual your output formatting must match this exactly for full credit. The file **FormatNumbers.py**, posted in Examples/pa6, may help achieve the column formatting. A set of seven matched pairs of input-output files are also posted in the same directory. The above pair is **in2-out2**. You'll also find the program **RandomInput.py** which will create a properly formatted, random input file for this project.

If you draw the above graph, you will find that it contains the 5-cycle $(5, 10, 15, 14, 9, 5)$, which we know requires 3 colors. Therefore the 3-coloring given in the above output file is optimal, and $\chi(G) = 3$ in this case. In general, you should not expect your output file to contain an optimal coloring, but only to establish the inequality $\chi(G) \leq k$, where k is the size of your color set. Not all of the output files given in Examples/pa6 are optimal, and for the larger ones, it is probably impossible to tell what $\chi(G)$ actually is.

Of all our programming projects thus far, this one perhaps affords you the greatest freedom in developing your own techniques. It is recommended that you begin by writing a program that always produces a proper coloring. Once that is done, experiment with different strategies to improve your results. As you experiment, you will likely find that your estimate of $\chi(G)$ improves on some examples (by getting smaller), and worsens on others.

Start this project as soon as possible and get help from TAs and course tutors as needed. Turn in your two files **graph.py** and **GraphColoring.py** to Gradescope before the due date.