```
1   Find_SCC: // from 3/14/19 lecture
2       DFSAll(G)
3       DFSAll(Gr)
4       return SCC_list // list SCCs with a list of vertices within each
5
6   Rainbow_Walk:
7       scc_list = Find_SCC
8       for each scc in scc_list:
9           color_list = [1..k]
10          for each vertex v within scc:
11              for each adjacent vertex u within scc:
12                  mark color of edge (v,u) in color_list
13          if color_list is completely marked:
14              return true
15      return false
```

```
Find_SCC: // from 3/14/19 lecture
    DFSAll(G)
    DFSAll(Gr)
    return SCC_list // list SCCs with a list of vertices within each

Rainbow_Walkpt2:
    arrange SCC_list into a DAG metagraph

    //returns a list of metagraph nodes in topologically sorted order
    Topological_meta_graph[] = run DFSall and DFSall reverse

    color_needed_list = [1..k]

    for(every metagraph node v in Topological_meta_graph[])
    {
        remove colors contained in Topological_meta_graph[v]
        if(color_needed_list.isempty)
            return true
    }
    return false
```

Worked with Jeremy Varghese.

25 (100 PTS.) Rainbow walk

We are given a directed graph with n vertices and m edges ($m \geq n$), where each edge e has a color $c(e)$ from $\{1, \ldots, k\}$.

25.A. (20 PTS.) Describe an algorithm, as fast as possible, to decide whether there exists a closed walk that uses all k colors. (In a walk, vertices and edges may be repeated. In a closed walk, we start and end at the same vertex.)

25.B. (80 PTS.) Now, assume that there are only 3 colors, i.e., $k = 3$. Describe an algorithm, as fast as possible, to decide whether there exists a walk that uses all 3 colors. (The start and end vertex may be different.)

25

Solution:

1. **Idea:** The idea behind this algorithm is to find all strongly connected components of the graph and then track which edge colors are between vertices in each SCC. The method to find strongly connected components is similar to the one we went over in the lecture on March 14th. By the definition of a strongly connected component there is a path between all pairs of vertices within each in our metagraph. Since each vertex can visit every node including itself we can create a closed path where it utilizes each edge between other nodes in the SCC ending at iself. From this we can track the colors of all the edges between nodes within the same SCC for all SCCs and then return true in the case a SCC utilizes each of the k colors.

   Psuedocode: See image 1.

   Analysis: The SCC algorithm is directly from lecture, utilizing two calls to DFSAll, giving a runtime of $O(m + n)$. Traversing through each list of SCC and tracking edge colors takes worse case $O(m)$. Since we are given the fact that m is larger than or equal to n in this question we can simplify our runtime down to a total $O(m)$.

2. **Idea:** The approach we took for this was to modify our part A algorithm. By contructing a meta graph, we create a new graph with groups of nodes. With each metagraph node, we have a list of colors that has all of the edges cover within that grouping of original nodes. We then compare this list against our master list of all colors needed. If the metagraph node contains all colors needed, we return true. If not, we traverse the next metagraph node in the topologically sorted list and see if it will complete our set of utilizing all k colors. If so, we return true. If not, we continue our traversal until all metagraph nodes are exhausted. If all are exhausted without completing the set of K colors, we return false.

   Psuedocode: See Image 2.

   Analysis: The SCC algorithm is directly from lecture, utilizing two calls to DFSAll, giving a runtime of $O(m + n)$. Traversing through each list of SCC and comparing edge colors also

takes $O(m)$ at worst case. If not all colors are satisfied by the metagraph node, traversing other metagraph nodes at worst case takes $O(n)$ time. Because these are called sequentially, the largest runtime is $O(m)$.
