CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)
Homework 8 Problem 1

Solution: (a) We construct a new graph $G' = (V', E')$ defined as follows and reduce the problem to a single search problem:

- $$V' = (V \times \{0, 1, 2, 3, 4\}) \cup \{(s, 0), t\} \text{ where } c(s) = c(t) = 0$$

  The vertices keep track of how many nodes with colors that are not 0 have been traversed. 4 is used to represent any number larger than 4 since it would be an invalid path. We also create a start node $(s, 0)$ that directs to all nodes in X, and a terminal node t that is the descendant of all nodes in Y.

- $$\begin{aligned} E' =& \{(u, i)(v, i) \mid uv \in E, i \in \{0, 1, 2, 3, 4\}, c(v) = 0\} \cup \\ & \{(u, i)(v, \max(i + 1, 4)) \mid uv \in E, i \in \{0, 1, 2, 3, 4\}, c(v) > 0\} \cup \\ & \{(s, 0)(v, 0) \mid v \in X, c(v) = 0\} \cup \\ & \{(s, 0)(v, 1) \mid v \in X, c(v) > 0\} \cup \\ & \{(u, i)(t) \mid u \in Y, i < 4\} \end{aligned}$$

  For every edge $uv \in E$, there are exactly 5 edges $(u, i)(v, j) \in E'$ with j dependent on i and $c(v)$. We can keep track of the number of nodes with colors not equal to 0 in this way. All nodes in Y that are reached in a path with at most 3 nodes with colors that are not 0 are connected to the terminal node t.

We do a whatever-first search using $(s, 0)$ as the start node on G' and see if there is a path to the terminal node t. If there is a path from $(s, 0)$ to t, then the algorithm outputs TRUE; otherwise, it outputs FALSE. Using the construction above, the number of vertices of G' is $|V'| = 5|V| + 2$ and the number of edges $|E'| = 5|E| + |X| + 4|Y| < 5|E| + 5|V|$. Therefore, this will take $O(|V'| + |E'|) = O(|V| + |E|) = O(m + n)$ time.

(b) To reduce the problem to (a), we construct a graph $G' = (V', E')$ defined as follows:

- $$V' = \{uv \mid uv \in E\}$$

  The vertices of G' are the edges of G.

- $$E' = \{(uv)(vw) \mid uv, vw \in V'\}$$

  There is an edge from vertex uv to vertex vw because these two edges share a common node v in G. Note that v must be the end vertex of one edge and the start vertex of the other edge.

Then we define X' and Y' as follows:

- $$X' = \{uv \mid uv \in V', u \in X\}$$

  If an edge is in the form uv, then it must start with u. Therefore, all edges with start node in X are in X'.

- $$Y' = \{uv \mid uv \in V', v \in Y\}$$

  If an edge is in the form uv, then it must end with v. Therefore, all edges with end node in Y are in Y'.

Since at most three edges in G with colors that are not 0 can be reached, at most three vertices in G' with colors that are not 0 can be reached. Then the problem is reduced to (a) since all edges are now uncolored and all vertices are colored. We pass the parameters $G', X', Y', c(e)$ where $e \in V'$ to the algorithm described in (a) and observe the output. The answer to the question in (b) is exactly equal to the output.

Time complexity analysis: The size of E' is equal to the sum of the product of in-degree and out-degree of each vertex in G because each in-edge and out-edge pair contributes to one edge in E', which is equal to the square of the sum of the out-degrees. Therefore, $|E'| = O(m^2)$. It is easy to see that $|V'| = m$. The construction takes $O(m^2 + n)$ time assuming $O(n)$ time to construct arrays to check if a node is in X or Y. The total time complexity of this algorithm is then $O(m^2 + n) + O(|V'| + |E'|) = O(m^2 + n)$.

■

CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)
Homework 8 Problem 2

Solution: (a) We can convert the original directed graph to a meta-graph, then use the property that a meta-graph is acyclic to solve the problem. Since a meta-graph is a DAG, it can be topologically sorted using DFS. In order for a node to be reachable by every other node, it must be at the last position of the topological sort; otherwise the SCC after it would not be able to reach it. To check if last SCC can be reached by all other SCCs, we reverse the meta-graph and run a single search and see if all SCCs are reached. If all SCCs are reached, then we compare k with the number of nodes in the last SCC in the topological sort. If k is larger, then we know there are not enough happy nodes and the algorithm outputs FALSE. Otherwise, the algorithm outputs TRUE. Following is the pseudocode for the algorithm.

```
ContainsHappyNodes(G, k):
  compute meta-graph G^SCC of G
  do DFS(G^SCC) and let S be the SCC with the smallest post-visit time
  mark all nodes in (G^SCC)^rev as unvisited
  do a single whatever-first search on (G^SCC)^rev using S as the start node
  if all nodes in (G^SCC)^rev are visited
      n ← number of nodes in S
      if n ≥ k
          return True
  return False
```

Time complexity analysis: Both computing meta-graph and DFS take $O(m+n)$ time according to lectures. Reversing G^{SCC} takes $O(m)$ time. The single whatever-first search on G^{SCC} takes $O(m+n)$ time since G^{SCC} cannot have more edges or vertices than G. Counting nodes in S takes $O(m+n)$ time with a single search. Overall, the time complexity is $O(m+n)$.

(b) We begin by computing the meta-graph and see if the meta-graph can be converted to a single SCC by creating cycles in the meta-graph and concatenating SCCs. We observe that the last edge added should be from the smallest SCC in the topological sort to the largest SCC to form a cycle that covers every SCC and thus converts the graph to a single SCC. Therefore, the problem is reduced to: *can we add at most one edge such that there is an edge between every continuous pair of SCCs in the topological sort?*
Since the added edge is part of the cycle, it must be true that the added edge is between two consecutive SCCs in the topological sort, otherwise the cycle would not cover all the SCCs. Therefore, we iterate through SCCs in topological order and see if there is an edge from the SCC to the next SCC. If there is not, then we add the edge and continue the iteration. If there is still a "discontinuity" in the remaining SCCs, then we return FALSE; otherwise, return TRUE. Following is the pseudocode for the algorithm.

```
StronglyConnectedWithTwoMoreEdges(G):
  compute meta-graph G^SCC of G
  do DFS(G^SCC) and output vertices in topological order
  edgeAdded ← False
  for each u in the computed order
      if u is not the last node
          v ← the node after u in the computed order
          if uv ∉ edge(G^SCC)
              if edgeAdded
                  return False
              edgeAdded ← True
  return True
```

Time complexity analysis: To check if an edge is in G^{SCC}, we can define a 2-d array *edgeCheck*$[n][n]$ where n is the number of vertices. *edgeCheck*$[i][j] = 1$ if ij is an edge in G^{SCC}, otherwise *edgeCheck*$[i][j] = 0$. This requires $O(m)$ time upfront and $O(1)$ time for retrieval. Since only constant-time operations are involved in the for-loop, the for-loop takes $O(n)$ time. Both DFS and computing meta-graph take $O(m + n)$ time. Therefore, the time complexity of the algorithm is $O(m + n)$.

■