Homework 3.
Due: Tuesday, June 22, 2021 before 11:59PM via Gradescope. Late submission with no penalty by Wednesday, June 23, 2021 before 11:59AM.

Problem 1 (Computopia).

Part 1.
Graphic Problem:
Let intersection be the vertices and streets be the edges, we are to design an algorithm to see if there is always a path between every pair of vertices.
Explanation:
The problem can be solved in linear time because we only need to run SCCs on the graph one time and SCCs runs in linear time.
If more than 1 strongly connected components exist, then the claim is false because the edges are directed one way between strongly connected components.
Otherwise, if there is only 1 strongly connected component, we know that the graph is cyclic thus all vertices can be reached.

Part 2.
Graphic Problem:
Let intersection be the vertices and streets be the edges, we are to design an algorithm given a vertex that returns true if the vertex can reach back taking any possible edges.
Design:
1. Consider the given graph and call it G, and call the given vertex v.
2. Run $SCCs(G)$, retrieve the metagraph G' of the strongly connected components of graph G.
3. Run $DFS(G'_i)$, which G'_i are all sets of strongly connected components.
4. If v is visited, run $Explore(G, v)$, let G'_v be the SCC that contains v.
5. Return false if any vertex visisted is not in the vertices set of G'_v, otherwise return true.

Correctness:
In order to reach back to the townhall node taking any possible edges, the SCC that contains the townhall node must to reach to any other SCC, making it a sink.
To see if the SCC is a sink, simply just do DFS on the SCC and if it reaches to other nodes that's not in the component, we know that the SCC is not a sink.
Runtime analysis:
$SCCs(G), (DFS(G'_i)$ and $Explore(G, v)$ all have runtime complexity $O(|V| + |E|)$. While checking if a vertex exist in a set is $O(1)$ using hashmap.
Thus, the runtime is dominated by $O(|V| + |E|)$.

Problem 2 (red and blue edges).

Part a.
Design:
1. Create new graph $G' = (V, E')$ which E' are set of red edges.
2. Run $Explore(G', v)$ to explore the subroutine in G' starting at v.
3. Return true if u is visited, otherwise if the call ends and u has not been visited, return false.

Correctnessn:
If we are looking for only edges that are red, we can ignore blue edges and apply simple exploring subroutine method to find the path.
In this case, if v reaches u successfully in a graph that only contains red edges, we can tell that there must be a pure red path between them, vice versa.
Runtime analysis:
Explore Subroutine algorithm runs at $O(|V| + |E|)$ time complexity so that is our runtime.

Part b.
Design:
1. Create new graph $G_{red} = (V, E_{red})$ and $G_{blue} = (V, E_{blue})$ which E_{red}/E_{blue} are set of red/blue edges.
2. Run $Explore(G_{red}, v)$ and $Explore(G_{blue}, u)$. Let visited vertices added in the two explore subroutine calls be v_1 and v_2.
3. Compare v_1 and v_2, if at least one vertex appears in both of the visited set, return true, otherwise return false.

Correctness:
Since the blue edges need to be after all red edges in the path, we know that there exist a pure blue path before reaching vertex u, and all the edges before that are red.
The idea is to divide it into two subroutine problems, one starting at v, looking for only red edges, and the other one starting at u, looking for only blue edges.
We know that if such path exists, then the two subroutine must meet each other at some vertex.
Therefore, we look at the two visited vertices set, and their intersection(s) are possible optioins where the red and blue route meet, giving us a legal path.
Runtime analysis:
Similarily, Explore Subroutine algorithm runs at $O(|V|+|E|)$ time complexity so that is our runtime.