1. Let $G = (V, E)$ be directed graph. A subset of edges are colored red and a subset are colored blue and the rest are not colored. Let $R \subset E$ be the set of red edges and $B \subset E$ be the set of blue edges. Describe an efficient algorithm that given G and two nodes $s, t \in V$ checks whether there is an s-t path in G that contains at most one red edge and at most one blue edge. Ideally your algorithm should run in $O(n+m)$ time where $n = |V|$ and $m = |E|$.

Solution: We first describe an algorithm to check whether there is an $s - t$ path in the input graph G that contains at most one red edge and at most blue edge such that any red edge precedes any blue edge in the path. We will then expand the idea to design an algorithm that checks existence of paths with at most one red edge and at most one blue edge without the added restriction.

For an input graph $G = (V, E)$ with a set of red edges $R \subset E$ and a set of blue edges $B \subset E$, and two nodes $s, t \in V$, we will construct a new graph $G' = (V', E')$ as follows. G' has three copies of G, namely $G^{(1)} = (V^{(1)}, E^{(1)})$, $G^{(2)} = (V^{(2)}, E^{(2)})$, and $G^{(3)} = (V^{(3)}, E^{(3)})$. We use superscripts to denote which copy the node belongs to, so that for a node $u \in V$, $u^{(1)}$ is in copy $G^{(1)}$, $u^{(2)}$ is in copy $G^{(2)}$, and $u^{(3)}$ is in copy $G^{(3)}$. From each copy we remove the red and blue edges. If a path wants to use a red or a blue edge, we will force it to go to a different copy. Note that for each edge $(u, v) \in E \setminus (R \cup B)$ (i.e. it is neither red nor blue), we have three edges $(u^{(1)}, v^{(1)}) \in E^{(1)}$, $(u^{(2)}, v^{(2)}) \in E^{(2)}$, and $(u^{(3)}, v^{(3)}) \in E^{(3)}$. For each red edge $(u, v) \in R$, we will create one edge $(u^{(1)}, v^{(2)})$ in G' (notice that the edge goes from the copy $G^{(1)}$ to the copy $G^{(2)}$). For each blue edge $(u, v) \in B$, we will create one edge $(u^{(2)}, v^{(3)})$ in G' (note again that the edge goes between two copies). We add a "super source" node s' with edges $(s', s^{(1)})$, $(s', s^{(2)})$, and $(s', s^{(3)})$ to G'. We also add a "super sink" node t' with edges $(t^{(1)}, t')$, $(t^{(2)}, t')$, and $(t^{(3)}, t')$ to G'. We claim that there exists an $s' - t'$ path in G' iff there exists an $s - t$ path in G that contains at most one red edge and at most blue edge such that any red edge precedes any blue edge in the path (a proof sketch for the final solution is provided later). This problem can then be solved in $O(|V'| + |E'|)$ using basic graph search. Note that $|V'| = O(|V|)$ and $|E'| = O(|E'|)$.

We can use the same idea to check whether there is an $s - t$ path in the input graph G that contains at most one red edge and at most blue edge such that any blue edge precedes any red edge in the path (note the reversal) by adding blue edges from copy $G^{(1)}$ to copy $G^{(2)}$ and red edges from $G^{(2)}$ to $G^{(3)}$. To handle both cases, we can create two more copies.

For the original problem, we have as input a graph $G = (V, E)$ with a set of red edges $R \subset E$ and a set of blue edges $B \subset E$, and two nodes $s, t \in V$. We will construct G' from five copies of G, namely $G^{(1)}$, $G^{(2)}$, $G^{(3)}$, $G^{(4)}$, and $G^{(5)}$. We remove the red and blue edges from each copy so that $E^{(i)} = \{(u^{(i)}, v^{(i)}) \mid (u, v) \in E \setminus (R \cup B)\}$, for $i \in \{1, 2, 3, 4, 5\}$. G' is then as follows.

$$
\begin{aligned}
V' &:= V^{(1)} \cup V^{(2)} \cup V^{(3)} \cup V^{(4)} \cup V^{(5)} \cup \{s', t'\} \\
E' &:= E^{(1)} \cup E^{(2)} \cup E^{(3)} \cup E^{(4)} \cup E^{(5)} \\
&\quad \cup \{(u^{(1)}, v^{(2)}) \mid (u, v) \in R\} \\
&\quad \cup \{(u^{(2)}, v^{(3)}) \mid (u, v) \in B\} \\
&\quad \cup \{(u^{(1)}, v^{(4)}) \mid (u, v) \in R\} \\
&\quad \cup \{(u^{(4)}, v^{(5)}) \mid (u, v) \in B\} \\
&\quad \cup \{(s', s^{(i)}) \mid i \in \{1, \ldots, 5\}\} \\
&\quad \cup \{(t^{(i)}, t') \mid i \in \{1, \ldots, 5\}\}
\end{aligned}
$$

We want to check if there is a $s' - t'$ path in G'. This can be done by using a basic graph search from s' and checking if t' is visited. Constructing the graph takes $O(|V'| + |E'|)$ time. The basic graph search can be done in $O(|V'| + |E'|)$ time. There are 5 copies of each vertex in V and 2 extra vertices s' and t'. Thus $V' = (5|V| + 2) = O(|V|)$. Similarly, each non-colored edge in $E \setminus (R \cup B)$ is copied 5 times. Each red or blue colored edge in $R \cup B$ contributes 2 edges to G'. Then there are 5 extra edges added for s' and 5 for t'. Thus $E' = (5|E \setminus (R \cup B)| + 2|R| + 2|B| + 10) = O(|E|)$. Therefore, we conclude that this algorithm takes $O(n + m)$ time.

It is left to prove the following claim.

Claim 1. *The graph G' (as constructed above) has an $s' - t'$ path iff the input graph G has an $s - t$ path that contains at most one red edge and at most one blue edge.*

We only provide a proof sketch for the above claim. For the forward direction, consider any $s' - t'$ path in G'. There are 5 cases to consider, but we will only consider one and the rest follow similarly. We consider the case where the $s' - t'$ path in G' starts in the copy $G^{(1)}$ and ends in copy $G^{(3)}$. So, by construction of G' the $s' - t'$ path is of the form $s', s^{(1)}, u_1^{(1)}, u_2^{(1)}, \ldots, u_j^{(1)}, u_{j+1}^{(2)}, u_{j+2}^{(2)}, \ldots, u_k^{(2)}, u_{k+1}^{(3)}, u_{k+1}^{(3)}, \ldots, u_\ell^{(3)}, t^{(3)}, t'$. Note that the path switches copies on the edges $(u_j^{(1)}, u_{j+1}^{(2)})$ and $(u_k^{(2)}, u_{k+1}^{(3)})$. These are red and blue edges respectively in G. The corresponding path in G is given by $s, u_1, u_2, \ldots, u_j, u_{j+1}, u_{j+2}, \ldots, u_k, u_{k+1}, u_{k+1}, \ldots, u_\ell, t$ uses only one red edge (u_j, u_{j+1}) and one blue edge (u_k, u_{k+1}).

For the other direction, there are again 5 cases to consider, but we will only consider one and the rest follow similarly. We consider the cases where the $s - t$ path in G uses one red edge and one blue edge such that the red edge precedes the blue edge. So the path is of the form $s, u_1, u_2, \ldots, u_j, u_{j+1}, u_{j+2}, \ldots, u_k, u_{k+1}, u_{k+1}, \ldots, u_\ell, t$ where (u_j, u_{j+1}) is the only red edge and (u_k, u_{k+1}) is the only blue edge. We can construct a corresponding path in G' given by $s', s^{(1)}, u_1^{(1)}, u_2^{(1)}, \ldots, u_j^{(1)}, u_{j+1}^{(2)}, u_{j+2}^{(2)}, \ldots, u_k^{(2)}, u_{k+1}^{(3)}, u_{k+1}^{(3)}, \ldots, u_\ell^{(3)}, t^{(3)}, t'$ where we have had to use the red edge $(u_j^{(1)}, u_{j+1}^{(2)})$ to switch copies from $G^{(1)}$ to $G^{(2)}$ and have had to use the blue edge $(u_k^{(2)}, u_{k+1}^{(3)})$ to switch copies from $G^{(2)}$ to $G^{(3)}$. ■

Rubric: 10 points:
- 3 for correct vertices and edges.
  - −1 for forgetting "directed".
- 1 for stating the correct problem.
  - "Breadth-first search" is not a problem; it's an algorithm.
- 2 points for correctly applying the correct algorithm.
- 2 points for justifying the reduction.
- 2 points for time analysis in terms of the input parameters.
  - −1 for not including the time to build the reduction.

2. The police department in the city of Shampoo-Banana has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. The city needs an algorithm to check whether the mayor's contention is indeed true.

   - Formulate this problem graph-theoretically, and describe an efficent algorithm for it.
   - Suppose it turns out that the mayor's original claim is false. Call an intersection u *good* if any intersection v that you can reach from u has the property that u can be reached from v. Now the mayor claims that over 95% of the intersections are good. Describe an efficient algorithm to verify her claim. Your algorithm should basically be able to find all the good intersections.

   Ideally your algorithms for both parts should run in linear time. You will receive partial credit for a polynomial-time algorithm.

Solution:

- *Graph Model:* Create a directed graph $G(V, E)$ as follows: (i) for each intersection i, include a vertex v_i in V, and (ii) for each one-way street from intersection i to intersection j, include a directed edge (v_i, v_j) in E.
- *Mayor's first claim*: There is a way to drive legally from any intersection in the city to any other intersection if and only if the graph G is strongly connected. Use the algorithm learned in class (Tarjan's algorithm) to find the meta-graph of strongly connected components corresponding to G. If there is only one vertex in the meta-graph (i.e., G is strongly connected), then Mayor's first claim is true; otherwise, it's false.

  Since Tarjan's algorithm takes linear time, we can verify the first claim in linear time.
- *Mayor's second claim*: Observe that an intersection i is good if and only if the vertex v_i corresponds to a sink vertex in the meta-graph. We first count the number of good intersections using the following algorithm.

```
CountGoodIntersection(G(V,E)):
  g ← 0    ⟨⟨number of good intersections⟩⟩
  construct the meta-graph of G. Let G^SCC denote the meta-graph.
  create array SCC from G^SCC where SCC[v] gives the name of the SCC containing vertex v.
  identify and mark all sink vertices of G^SCC
  for each vertex v ∈ V,
      if SCC[v] is marked
          g ← g + 1
  return g
```

  Then, mayor's second claim is true if and only if CountGoodIntersection$(G(V, E)) > 0.95|V|$.

  Since the construction of the meta-graph is linear, and the for-loop also takes linear time, we can verify mayor's second claim in linear time.

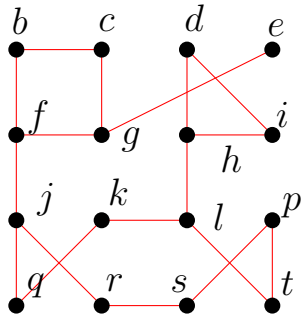
■

Rubric: Out of 10 points, we allocate:

- **+1 points** for the graph model.
- **+3 points** for verifying the first claim.
  - **3 points max** for linear algorithm.
  - **2 points max** for polynomial-time algorithm.
  - **2 points off** for major mistakes.
  - **1 point off** for minor mistakes.
- **+6 points** for verifying the second claim.
  - **6 points max** for linear algorithm.
  - **4 points max** for polynomial-time algorithm.
  - **4 points off** for major mistakes.
  - **2 point off** for minor mistakes.

3. Given an undirected connected graph $G = (V, E)$ an edge (u, v) is called a cut edge or a bridge if removing it from G results in two connected components (which means that u is in one component and v in the other). The goal in this problem is to design an efficient algorithm to find *all* the cut-edges of a graph.

   - What are the cut-edges in the graph shown in the figure?



     Solution: (e, g), (f, j), and (h, l) are the cut-edges. ■

   - Given G and edge $e = (u, v)$ describe a linear-time algorithm that checks whether e is a cut-edge or not. What is the running time to find all cut-edges by trying your algorithm for each edge? No proofs necessary for this part.

     Solution: To see whether $e = (u, v)$ is a cut-edge, we can remove e from G and use any basic search algorithm to determine whether u is still connected to v. If it is, then e is not a cut-edge, otherwise e is a cut-edge.

     It takes $O(n + m)$ time to perform the basic search starting at u. If we perform this once for each of the m edges, the algorithm takes $O(m^2)$ time. ■

   - Consider *any* spanning tree T for G. Prove that every cut-edge must belong to T. Conclude that there can be at most $(n - 1)$ cut-edges in a given graph. How does this information improve the algorithm to find all cut-edges from the one in the previous step?

     Solution: For any edge $e = (u, v) \notin T$, removing e would not cause the graph to be disconnected, because between any two vertices in G, there is a path in the spanning tree T that connects them.

     Because a spanning tree has $n - 1$ edges and only edges in the spanning tree can be cut-edges, there can be at most $n - 1$ cut-edges in a given graph.

     The number of candidate cut-edges drops from m to $n - 1$. If the algorithm performs the check for each of the $n - 1$ edges, it takes $O(mn)$ time. ■

   - Suppose T is a spanning tree of G rooted at r. Prove that an edge (u, v) in T where u is the parent of v is a cut-edge iff there is no edge in G with one end point in T_v (sub-tree of T rooted at v) and one end point outside T_v.

     Solution: ($\Longrightarrow$): By contrapositive: Suppose there is an edge e in G with one endpoint in T_v and one endpoint outside of T_v. Since T is a tree, removing (u, v) from T leaves it in two connected components: one that contains u and another that contains v (which in fact is T_v). The edge e will connect these two components to create a single connected component in with no cycles. Thus the graph T' obtained from T by removing the edge (u, v) and adding the edge e, is a spanning tree of G. Since (u, v) is not on T', it cannot be a cut-edge.

     ($\Longleftarrow$): Consider removing the edge (u, v) from T. This gives us two connected components: one containing u and T_v. Since T_v does not have any edges leaving it in

$G-(u,v)$, it is its own connected component in $G-(u,v)$, and hence u is in another connected component of $G-(u,v)$. Thus (u,v) is a cut-edge. ■

- Use the property in the preceding part to design a linear-time algorithm that outputs all the cut-edges of G. You don't have to prove the correctness of the algorithm but you should point out how your algorithm ensures the desired property. *Hint:* Consider a DFS tree T and some additional information you can compute during DFS. You may want to run DFS on the example graph with the cut edges identified.

Solution: Let T be a DFS tree. We denote the previsit time of a vertex u in T by $pre(u)$. For each node u define:

$$low(u) = \min \begin{cases} pre(u) \\ pre(w) \text{ where } (v,w) \text{ is a back edge for } v \in T_u. \end{cases}$$

To clarify, we remark that $low(u)$ will be $pre(u)$ if there is not back edge goint out of T_u. The following preprocessing procedure is a modification of DFS that for each u, records the previsit time $pre(u)$, the predecessor vertex $pred(u)$, and the value $low(u)$.

```
LowDFS(G):
  for all u ∈ V
      pred(u) ← NULL
  pick an arbitrary vertex u ∈ V
  LowDFSAux(G, u)
```

```
LowDFSAux(G, u):
  mark u as a visited vertex
  time ← time + 1
  pre(u) ← time
  low(u) ← pre(u)
  for each edge (u, v) in Adj(u)
      if v is not marked as visited
          pred(v) ← u
          LowDFSAux(G, v)
          low(u) ← min{low(u), low(v)}
      else if v ≠ pred(u)
          low(u) ← min{low(u), pre(v)}
```

Then the following procedure will return the set of all of the cut-edges:

```
AllCutEdges(G):
  pre(), low(), pred() ← LowDFS(G)
  S ← ∅
  for each vertex v ∈ V
      if low(v) = pre(v) and pred(v) ≠ NULL
          add (pred(v), v) to S
  return S
```

The preprocessing procedure is a DFS with predecessor and previsit time, with additional constant amount of work at every step in order to calculate $low(u)$. Thus the running time of the algorithm is exactly the same as DFS: $O(m+n)$. Furthermore, AllCutEdges(G) runs in $O(n)$ time, so the total running time is $\boldsymbol{O(m+n)}$, or linear in the size of the input.

Proof of Correctness: Since the modification to the DFS algorithm only adds a step recording $low(u)$, it does not affect any other operations, so the tree produced is still a DFS tree.

Furthermore, $low(u)$ is calculated correctly: By induction.

- *Base Case*: When u has no children then $low(u)$ is calculated correctly: either $low(u) = pre(u)$ or there are back edges from u, in which case the assignment in the "else" clause correctly calculates the min $\{pre(w) : (u, w)$ is a back edge$\}$.
- *Inductive hypothesis*: Assume that $low(v)$ is correctly computed for all children v of u.
- *Inductive Step*: If there are no back edges leaving T_u then $low(u) = pre(u)$. If $low(u)$ is achieved by $pre(w)$ where (u, w) is a back edge, then again the "else" clause will correctly compute $low(u)$. Finally, if $low(u)$ is achieved by $pre(w)$ where (v, w) is a back edge for $v \in T_u$, $v \notin u$, then the line after the recursive call LowDFSAux(G, v) will correctly compute $low(u)$ by the inductive hypothesis.

Finally, we will prove that the cut-edges are exactly the edges (u, v) where u is a parent of v and $low(v) = pre(v)$.

Recall a DFS tree is a spanning tree, so from the previous part we know that in T, (u, v) (where u is the parent of v) is a cut-edge iff there is no edge in G with one end point in T_v and one end point outside T_v other than (u, v).

Since T is a DFS tree, an edge that is not (u, v) with one end point in T_v and one end point outside of T_v must be a back-edge from a vertex in T_v to some w such that $pre(w) < pre(v)$ (recall that there are no cross-edges in a DFS tree). Thus, such an edge exists iff $low(v) < pre(v)$, so we conclude that (u, v) is a cut-edge iff $low(v) = pre(v)$. □

Alternate Solution: Given an arbitrary rooted spanning tree T, do a post-order numbering of the vertices according to T. For each node v the numbers of vertices in T_v will be in an interval $[a_v, b_v]$. For each node v, we keep track of the smallest-numbered vertex and the largest-numbered vertex that an edge leaving T_v can reach. This information can be updated and will allow for checking the desired condition. ■

Rubric: Out of 10 points, we would allocate:
- 1 point for part (a)
- 2 points for part (b)
- 2 points for part (c)
- 2 points for part (d), 1 per direction
- 3 points for part (e)