1. **Solution:** (a) 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 vertex and at most blue vertex such that any red vertex precedes any blue vertex in the path. Our algorithm will indeed look for an $s-t$ walk with such constraints, rather than a path, but one can see that these two problems are equivalent. We will then expand the idea to design an algorithm that checks the existence of paths with at most one red vertex and at most one blue vertex without the added restriction.

For an input graph $G = (V, E)$ with a set of red vertices $R \subset V$ and a set of blue vertices $B \subset V$, and two nodes $s, t \in V \setminus (R \cup B)$, we will construct a new graph $G' = (V', E')$ as follows. The graph 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 $u \in V$, the vertex $u^{(1)}$ is in the copy $G^{(1)}$, the vertex $u^{(2)}$ is in the copy $G^{(2)}$, and the vertex $u^{(3)}$ is in the copy $G^{(3)}$. From each copy we remove the edges going out of the red and blue vertices. If a path wants to use a red or a blue vertex, we will force it to go to a different copy. Note that for each edge $(u, v) \in E$ where $u \in V \setminus (R \cup B)$ we have $(u^{(1)}, v^{(1)}) \in E^{(1)}$, $(u^{(2)}, v^{(2)}) \in E^{(2)}$, and $(u^{(3)}, v^{(3)}) \in E^{(3)}$. For each edge $(u, v) \in E$ outgoing a red vertex $u \in R$, 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 edge $(u, v) \in E$ outgoing a blue vertex $u \in B$, 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 vertex and at most blue vertex such that any red vertex precedes any blue vertex 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 vertex and at most blue vertex such that any blue vertex precedes any red vertex in the path (note the reversal) by adding edges outgoing the blue vertices from copy $G^{(1)}$ to copy $G^{(2)}$ and edges outgoing the red vertices 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 vertices $R \subset V$ and a set of blue vertices $B \subset V$, and two nodes $s, t \in V \setminus (R \cup V)$. We will construct G' from five copies of G, namely $G^{(1)}$, $G^{(2)}$, $G^{(3)}$, $G^{(4)}$, and $G^{(5)}$. We remove the edges outgoing the red and blue vertices from each copy so that $E^{(i)} = \{(u^{(i)}, v^{(i)}) \mid (u, v) \in E, u \in V \setminus (R \cup B)\}$, for $i \in \{1, 2, 3, 4, 5\}$. G' is then as follows.

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

We want to check if there is an $s'-t'$ 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 edge outgoing a non-colored vertex in $V \setminus (R \cup B)$ is copied 5 times. Each edge outgoing a red or blue colored vertex in $R \cup B$ contributes 2 edges to G'. Then there are 5 extra edges added for s' and 5 for t'. Thus $|E'| \leq (5|E| + 10) = O(|E|)$. Therefore, we conclude that this algorithm takes $O(|V| + |E|)$ 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 vertex and at most one blue vertex.*

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 the 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 edges outgoing red and blue vertices respectively in G. The corresponding walk 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 vertex u_j and one blue vertex u_k.

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 vertex and one blue vertex such that the red vertex precedes the blue vertex. So the walk is of the form $s, u_1, u_2, \ldots, u_j, u_{j+1}, u_{j+2}, \ldots, u_k, u_{k+1}, u_{k+2}, \ldots, u_\ell, t$ where u_j is the only red vertex and u_k is the only blue vertex. 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 edge $(u_j^{(1)}, u_{j+1}^{(2)})$ to switch copies from $G^{(1)}$ to $G^{(2)}$ and have had to use the edge $(u_k^{(2)}, u_{k+1}^{(3)})$ to switch copies from $G^{(2)}$ to $G^{(3)}$. Note that these two edges are in G' because u_j is a red vertex and u_k is a blue vertex.

(b) Given an edge-colored graph G with red edges $R \subset E$ and blue edges $B \subset E$, we create a new graph G' by subdividing each edge $(u, v) \in E$ by introducing a new vertex x_{uv} and replacing the edge (u, v) with (u, x_{uv}) and (x_{uv}, v). We then define the introduced vertices to have the same color as the edge they subdivided. That is, we define $R' := \{x_{uv} \mid (u, v) \in R\}$ and $B' := \{x_{uv} \mid (u, v) \in B\}$.

For every $s - t$ path P in G, let P' be the path in G' that replaces each edge of P with the two edges that represent it in G'. Observe that every path P in G visits the same number of red and blue edges as the number of red and blue vertices that the corresponding path P' visits in G', and vice versa. Therefore, there is an $s - t$ path P in G with at most one red edge and at most one blue edge, if and only if, there is an $s - t$ path P' in G' that visits at most one red vertex and at most one blue vertex. Therefore, solving the original problem in G is reduced to the vertex-version problem in G'. The size of the new graph is $|V'| = |V| + |E|$ and $|E'| = 2|E|$. Therefore, the new graph can be constructed in time $O(|V'| + |E'|) = O(|V| + 3|E|) = O(|V| + |E|)$. Then the algorithm from the previous part is run on the graph G' which takes $O(|V'| + |E'|) = O(|V| + |E|)$. Thus, the running time of this reduction is linear in the

size of the input graph.

■

Rubric:

(a) 6 points:

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

(b) 4 points:

- 2 points for the correct construction of G'
- 1 point for the correctness idea
- 1 point for the run time analysis

2.

- **Solution:** Begin by finding all strongly connected components in G. Union all components with two or more vertices. This union is the set of all vertices in at least one cycle. In other words

$$S = \{v \in W \mid W \subseteq V \text{is a strongly connected component in G with size} \geq 2\}$$

The algorithm for finding strongly connected components takes $O(V+E)$ time and the time it takes to union all SCC's with size ≥ 2 is $O(V)$, resulting in a total running time of $O(V+E)$. To see the correctness of the algorithm we observe that v is in a cycle iff the strongly connected component containing v contains a vertex other than v in it. We will sketch both directions. Suppose v is in a cycle and let u be a vertex in the cycle other than v. We can see that v can reach u and u can reach v which implies that both are in the same strongly connected component (includes also all the other vertices in the cycle and perhaps other vertices too). For the other direction suppose v is in a strongly connected component with u. Consider a path P from v to u and a path Q from u to v. We claim that the union of P and Q contains a cycle C that contains v. This is clear if P and Q do not intersect in any vertices other than u, v. If they intersect one needs a more careful argument that we leave for you to figure out. Alternatively, you can do a DFS from v and argue that v will discover all the vertices in P and Q since they are all reachable from v. This also implies that there will be back edge to v during DFS and will create a cycle that contains v — this is not a fully formal argument. ■

- **Solution:** First if G is a DAG. Iterate over the vertices and look for a vertex v with no in-edges. If one can be found, perform BFS/DFS starting from v. If all vertices are found, then v is the vertex we are looking for, otherwise, there are no vertices that have paths to all other vertices. The algorithm works by taking advantage of the fact

that each DAG will have at least one source, we find a source and check if it has a path to all other vertices. As a source has no in-edges, if it doesn't have a path to all other vertices, then no other vertex will either, because they will lack a path to the source. An alternate solution is to count the number of sources. If there is only one, then it has a path to all other vertices. If there are more than one, then no such vertex exists.

If G is not a DAG, first computer the meta-graph of the strongly connected components in linear time using the algorithm from class. This meta-graph will be a DAG. Now run the earlier algorithm on this meta-graph. If there is no vertex in the meta-graph that can reach all other vertices in the meta-graph, then there is also no vertex in the original graph that can do so. If there is a vertex in the meta-graph that can reach all other vertices in the meta-graph, then any element of that meta-vertex can reach all other vertices.

The algorithm for meta-graph construction takes $O(V+E)$ time, iterating over the vertices looking for one without in-edges takes $O(V+E)$ time, and checking if the found vertex reaches all vertices takes $O(V+E)$, resulting in a total running time of $O(V+E)$ time. ■

Rubric: 10 points

- Part one: 4 points
  - 1 for implementing or modifying an existing graph algorithm (ie. SCC) instead of using it as a black box.
  - 1 for missing each specific edge case (eg. not recognizing SCC's of size 1 have no cycles).
  - 2 for a more than linear time algorithm
  - 3 for an incorrect algorithm
  - * It is possible to lose all 4 points if the given algorithm is incorrect and tries to modify or implement an algorithm instead of using it as a black box.
- Part two: 6 points
  - 1 for implementing or modifying an existing graph algorithm (ie. meta-graph construction) instead of using it as a black box.
  - 1 for missing each specific edge case.
  - 2 for a more than linear time algorithm
  - 2 for finding an algorithm when G is a DAG but not generalizing the algorithm to all directed graphs.
  - 5 for an incorrect algorithm
  - * It is possible to lose all 6 points if the given algorithm is incorrect and tries to modify or implement an algorithm instead of using it as a black box.

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

- 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.

- ($\Longrightarrow$): 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.

- 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
          LowDFSAux(G, v)
          low(u) ← min{low(u), low(v)}
          pred(v) ← u
      else
          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:
- 0.5 for one missing cut edge, [-]1 for more than one missing cut edges
- 1.5 missing/incorrect algorithm, [-]0.5 missing/incorrect time analysis
- 1 incorrect proof, [-]1 missing/incorrect time analysis of the algorithm of part (b)
- 1 missing/incorrect proof for each direction (if and only if)
- 2 missing/incorrect algorithm, [-]0.5 missing description of modification to DFS, [-]0.5 missing/incorrect time analysis