A2 S2021

1. [similar to PQ2, just different state and different state transitions]
We represent this as a graph problem as follows. Each state is represented by the location of the two tokens. There is an edge between two vertices (states) if one of the tokens can be moved from state 1 to 2. To solve the problem, we run BFS on the graph from the starting state to a state where either token A or B is on the target.

2. [Tutorial 3 problem]
We use a modified BFS to solve this problem. Instead of only keeping track of the current vertex, we also keep track of the colours of the last two edges in our state. The rest of the BFS algorithm is the same, except when checking neighbours we make sure we don't have a walk with 3 consecutive edges of the same colour.

```
function redBluePaths(graph, s, t) {
    init queue, visited
    queue.push((s, null, null))

    while (!queue.empty())
        current, prevColor, prevPrevColor = queue.pop()

        for neighbour of current
            if (prevColor == prevPrevColor == current->neighbour.color) continue

            if (neighbour == t) return s-t path

            nextState = (neigbour, current->neighbour.color, prevColor)
            if (nextState not in visited)
                visited.add(nextState)
                queue.push(nextState)
    return null
}
```

Runtime: The runtime of this algorithm is the same as BFS, since we only added color to our state. Hence the runtime is O(|V| + |E|).

3. We assume that G is connected. We claim that if a directed acyclic graph contains a vertex from which all other vertices are reachable, then the vertex with the largest finish time in a DFS must be one of them. We prove this statement. Suppose the graph contains a vertex from which all other vertices are reachable, u. Suppose for sake of contradiction that the vertex with largest finish time is not u, but instead some vertex v where not all other vertices are reachable. If start[u] < start[v], then v must be a descendant of u, since all other vertices are reachable from u. Then by the parenthesis property, finish[u] > finish[v], which is a contradiction. If start[v] < start[u], then v was visited before u. If u is a descendant of v, then all other vertices must be reachable from v, since they are from u, which is a contradiction. If v is not a descendant of u, then we must have finish[v] < finish[u] by the parenthesis property, which is a contradiction. Hence we must have that if the graph contains a vertex from which all other vertices are

reachable, then the vertex with the largest finish time in a DFS must be one of them.

Our algorithm then works as follows. We first find all the strongly connected components of G, and contract each into a single vertex, so we have a directed acyclic graph of SCCs. We then run DFS on this DAG, and find the vertex with the largest finish time. We run BFS/DFS starting from this vertex to check if all other vertices are reachable from it. If so, we return any of the vertices in this SCC. Otherwise we return false.

Runtime: It takes O(m+n) time to find the SCCs of G. We then run two iterations of DFS, which take O(m+n) time each. Hence our algorithm runs in O(m+n) time.

4. [A2 problem]
We model the problem as follows: given an undirected graph (each vertex is an intersection, and each edge is a two-way street), convert it into a strongly connected directed graph (each vertex is an intersection, each edge is a one-way street, and each vertex is reachable from every other vertex in the graph). If the graph is not connected, it is obviously not possible, so we assume the graph is connected.

Our algorithm operates as follows. We first check if G contains any cut edges using the algorithm specified in the tutorials. If G contains a cut edge, we return that it is not possible to convert. Otherwise, we convert it into a strongly connected directed graph as follows.

From above, we have a DFS tree of G, rooted at some vertex s. We set the direction of all tree edges away from s (pointing down the tree). We set the direction of all non-tree edges towards s (up the tree, ie back edges). There are no cross edges, since we did DFS on an undirected graph, so all edges incident to a vertex were explored. We then return this new directed graph.

Correctness: If there is a cut edge uv in the graph, if we assign it a direction u->v, then u must be unreachable from v. We now show that our algorithm for generating the directed graph creates a strongly connected one. We have that G is strongly connected if and only if every vertex v is reachable from s (the root), and s is reachable from every vertex v. Every vertex is reachable from the root by construction. For every tree edge uv, there exists a directed path from v to u, and then, the general claim follows by induction. Since uv is not a cut edge, then low[v] <= start[u]. This means that there exists a back edge from some vertex b in the subtree of v to u or some ancestor a of u. Then the path v -> b -> a -> u is a directed path from v to u.

Runtime: Our algorithm for finding cut edges takes O(n + m) time. The algorithm for converting the undirected graph into a strongly connected directed graph runs DFS once, which takes O(n + m) time. Setting the direction of each edge takes O(m) time. Hence our algorithm runs in O(n + m) time.