Exercises - Graph Introduction Tutorial

1. **Prove the White-Path Theorem:** Recall the White-Path Theorem: *In a depth-first-forest of a Graph G, vertex v is a descendant of vertex u if and only if at the time that is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices.* Prove that the White-Path Theorem is correct.

   Solution:
   See the lecture note (Theorem 2).

2. **Prove the correctness of topological sort:** Recall the topological sort algorithm introduced in the lecture that runs a DFS and returns the vertices in descending order of their finished time. Show that the algorithm correctly sorts the vertices in a topological order.

   Solution:
   To prove the correctness of the topological sort algorithm, one should show that for any edge (u, v), u must appear before v.

   For the rest of the proof, see the lecture note (Theorem 4).

3. **Semi-connected graph:** A directed graph $G = (V, E)$ is semi-connected iff for each pair u and v of V, there is a path from u to v or there is a path from v to u.

   (a) Assume G has two vertices u and v, both of which have in-degree 0. Prove that G is not semi-connected.

   (b) Assume G has two vertices u and v, both of which have in-degree 1 and assume (w, u) and (w, v) are two arrows in G where $w \in V$ has in-degree 0 using 3a. Prove that G is not semi-connected.

   (c) Assume G is a DAG. Design an $O(|V|^2)$-time algorithm to decide whether G is semi-connected or not using 3a and 3b.

   (d) For arbitrary directed graph G, design an $O(|V|^2)$-time algorithm to decide whether G is semi-connected or not (use 3c).

   (e) Let $\mathcal{C}$ be the set of the strongly connected components of G. Let $G^* = (V^*, E^*)$ be the directed graph such that there is a corresponding vertex $v_C \in V^*$ for each component $C \in \mathcal{C}$, and $(v_C, v_{C'}) \in E^*$ if there is a path from a vertex in C to a vertex in C'. Obtain the sorted vertices $(v_1, v_2, \ldots)$ by running topological sort on G^*. Prove that $(v_i, v_{i+1}) \in E^*$ for $1 \leq i < |V^*|$ iff G is semi-connected. Use this property to design a faster algorithm that decides if G is semi-connected.

   Solution:

   (a) If G is semi-connected, there must exist a path from u to v or a path from v to u. However, u has in-degree 0, which means there is no path ending with u. The same argument applies on v, i.e., no path ends with v. Overall, there is no path from u to v and no path from v to u, and thus G is not semi-connected.

(b) Although w has in-degree 0 and no path ends with w, there might exist several paths starting from w and reaching all other vertices. So we cannot decide if the graph is semi-connected by looking at only one zero-in-degree vertex. In this case, we remove w and its arrows, and see if the rest of the graph is semi-connected or not. By the result of 3a, the rest of the graph is not semi-connected, and thus the original graph is also not semi-connected.

(c) First, we observed that a Dag always has a vertex with in-degree 0. Otherwise, there is a cycle in the graph, and it contradicts to the fact that the graph is acyclic.

Now, we design an algorithm to decide if G is semi-connected. Given a DAG G, if G has two or more vertices with in-degree 0, then G is not semi-connected (by the result of 3a). Otherwise, G has exactly one vertex with in-degree 0. We remove the vertex and repeat checking the number of vertices with in-degree 0 (see the result of 3b). Each check for the number of vertices with in-degree 0 consumes $O(|V|)$ time. There are $O(|V|)$ checks in total as we remove one vertex in each round. Thus the total time complexity is $O(|V|^2)$.

(d) By running the algorithm for finding strongly connected components on G, we obtain the strongly connected components of G. Any such component is semi-connected since the component is strongly connected. The components form a DAG if we draw all the arrows from component C to C' where there is a path from a vertex in C to a vertex in C'. We decide if the DAG is semi-connected or not by the algorithm of 3c. If the DAG is semi-connected, then G is also semi-connected. Otherwise, G is not semi-connected. This is because if there is a path from C to C', then any vertex in C can reach any vertex in C' (by the property of strong connectivity). By the result of 3c, the algorithm runs in $O(|V|^2)$.

(e) Suppose that for all $1 \leq i < |V^*|$, $(v_i, v_{i+1}) \in E^*$. Then, v_1 can reach all the rest of the components v_i for $i > 1$. By the property of strong connectivity, there exists a vertex u in component v_1 that can reach all the vertices in v_1. This means u can reach all the components v_i for $i > 1$. Since all v_i's are strongly connected, u can reach all the vertices in v_i. Thus the original graph G is semi-connected.

Conversely, suppose that there exists an i such that $(v_i, v_{i+1}) \notin E^*$, then v_i cannot reach v_{i+1} as explained below. Recall in topological sort, all the arrows are from left to right, and there is no back edge. Component v_i may reach v_j for some $j > i + 1$, but v_j cannot reach v_{i+1} since there is no back edge. Thus, v_i cannot reach v_{i+1} and the original graph G is also not semi-connected.

For the algorithm, we construct G^* as described in the question. The algorithm checks if $(v_i, v_{i+1}) \in E^*$ for $1 \leq i < |V^*|$. If so, returns that G is semi-connected. Otherwise, returns that G is not semi-connected. The algorithm only invokes the algorithms for strongly connected components and topological sort. Thus the running time is $O(|V| + |E|)$.

4. **Rolling Die Game:** Consider rolling a die on a grid plane, where each entry on the plane may be labeled "forbidden" at the beginning. Meanwhile, the bottom-left and top-right entries are not forbidden, but with labels s and t, respectively, where $s, t \in \{1, 2, 3, 4, 5, 6\}$. The die is initially put at the bottom-left entry with the side s up. At any round, the die can be rolled up or rolled right to any neighboring entries that are not labeled as forbidden. The goal is to decide if it is possible to move the die by rolling it up or right such that it is eventually placed at the top-right corner with the side t up.

Model the rolling die game as a graph problem and describe how to solve it. Note that you can make the decision on how the die was placed in the first place, as long as it has the side s up.

Solution:

One of the possible modeling: First, we use a 2-tuple (u, r) to describe the *configuration* of the die with u-side up and r-side on the right. In total, there are $6 \cdot 4$ possible configurations of the die. Next, we use a directed *rolling graph* $R = (V_R, E_R)$ to describe the change of die configurations after rolling. For each configuration (u, r), there is a vertex $v_{ur} \in V_R$. For any two vertices v_{ur} and $v_{u'r'}$ in V_R, there is an edge $(v_{ur}, v_{u'r'}) \in E_R$ with a label *right* if rolling

the die with configuration (u, r) to the right results in the configuration (u', r'). For example, (v_{12}, v_{51}) is such an edge with a label *right*. Similarly, there are edges $(v_{ur}, v_{u'r'}) \in E_R$ with a label *up* if rolling the die with configuration (u, r) up results in the configuration (u', v'). (Ex: (v_{12}, v_{42}) is such an edge with a label *up*.)

We also model the grid in a directed graph $G = (V, E)$, where each non-forbidden entry has a vertex in V. For two entries u and v, there is an edge $(u, v) \in E$ with a label *right* if v is directly to the right of u. Similarly, if the entry v is directly above the entry u, there is an edge $(u, v) \in E$ with a label *up*.

To decide if it is possible to solve the game, for each vertex in V, we maintain a list of possible die configurations. Specifically, the list of the possible die configurations of the bottom-left grid contains all 4 configurations with the side s up. Consider each plane entry $v \in V$ and its list of possible die configurations. If $(v, u) \in E$ has the label *right* (resp. *up*), then for any v's die configuration (u, r) (with corresponding $v_{ur} \in V_R$), add the configuration (u', r') (with corresponding $v_{u'r'} \in V_R$) to the u's list of possible die configurations if $(a_{xy}, a_{x'y'}) \in E_R$ has a label *right* (resp. *up*). Finally, if the top-right entry of the grid has a die configuration with the side t up, then the game can be solved.