DFS Edge Classification

The edges we traverse as we execute a depth-first search can be classified into four edge types. During a DFS execution, the classification of edge \((u, v)\), the edge from vertex \(u\) to vertex \(v\), depends on whether we have visited \(v\) before in the DFS and if so, the relationship between \(u\) and \(v\).

1. If \(v\) is visited for the first time as we traverse the edge \((u, v)\), then the edge is a tree edge.

2. Else, \(v\) has already been visited:
   
   (a) If \(v\) is an ancestor of \(u\), then edge \((u, v)\) is a back edge.
   
   (b) Else, if \(v\) is a descendant of \(u\), then edge \((u, v)\) is a forward edge.
   
   (c) Else, if \(v\) is neither an ancestor or descendant of \(u\), then edge \((u, v)\) is a cross edge.

After executing DFS on graph \(G\), every edge in \(G\) can be classified as one of these four edge types. We can use edge type information to learn some things about \(G\). For example, tree edges form trees containing each vertex DFS visited in \(G\). Also, \(G\) has a cycle if and only if DFS finds at least one back edge. Note that undirected graphs cannot contain forward edges and cross edges, since in those cases, the edge \((v, u)\) would have already been traversed during DFS before we reach \(u\) and try to visit \(v\).

Connected Components

A connected component is defined as a subgraph where there exists a path between any two vertices in it. Graph \(G\) is made up of separate connected components and it may be useful to be able to classify each vertex by which connected component it belongs to.
For undirected graph $G$, executing a BFS or DFS starting from a vertex $v$ will visit every other vertex in the same connected component as $v$. We can mark every vertex visited from a BFS/DFS from $v$ as being “owned” by $v$. As we iterate through all the vertices, we execute a BFS/DFS starting from a vertex if it has no owner (i.e. it is part of an undiscovered connected component) and mark all the vertices visited in that BFS/DFS. After iterating through all the vertices, each vertex will be marked by its owner, representing which connected component it is a part of. In summary, the algorithm is the following:

1. For each vertex $v$ in undirected graph $G$
   (a) If $v$ has no owner, it is part of an undiscovered connected component. Execute BFS or DFS starting from $v$ and mark all the vertices as being owned by $v$
   (b) Else, if $v$ has an owner, it is part of a connected component we’ve already discovered. Ignore $v$ and move on to the next vertex.

The runtime of this algorithm is $O(|V| + |E|)$ since each vertex is visited twice (once by iterating through it in the outer loop, another by visiting it in BFS/DFS) and each edge is visited once (in BFS/DFS).

**Strongly Connected Components**

The algorithm above does not work with directed graphs. For undirected graphs, finding a path from $u$ to $v$ implies that there exists a path from $v$ to $u$. This is not the case for directed graphs. We can still separate the directed graphs into strongly connected components, which are components
in directed graphs where any two vertices has a path in between each other. Note that this is the same definition as connected components above, but applied to directed graphs.

The intuition that will help us separate a directed graph into strongly connected components is realizing that a strongly connected component with its edges’ directions reversed is still a strongly connected component. We will introduce \( G^T \), which is the transpose of directed graph \( G \). \( G^T \) and \( G \) are the same graph except the edge directions are reversed in \( G^T \), i.e. if edge \((u, v)\) is in \( G \), then the edge \((v, u)\) is in \( G^T \). An algorithm to find strongly connected components goes as follows:

1. Execute DFS on \( G \) (starting at an arbitrary starting vertex), keeping track of the finishing times of all vertices
2. Compute the transpose, \( G^T \)
3. Execute DFS on \( G^T \), starting at the vertex with the latest finishing time, forming a tree rooted at that vertex. Once a tree is completed, move on to the unvisited vertex with the next latest finishing time and form another tree using DFS and repeat until all the vertices in \( G^T \) are visited
4. Output the vertices in each tree formed by the second DFS as a separate strongly connected component

We can reduce a directed graph \( G \) to a graph of its strongly connected components, as seen above. Note that the graph of \( G \)'s strongly connected components cannot contain a cycle, since a cycle of strongly connected components can itself be reduced into a single strongly connected component. We call a directed graph with no cycles a dag, short for directed acyclic graph. We can thus say that every directed graph \( G \) can be reduced to a dag of its strongly connected components.