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**. These type of edges result when \(u\) and \(v\) are reached by different branches.

![Diagram showing tree edge, back edge, forward edge, and cross edge]

After executing DFS on graph \(G\), every edge in \(G\) can be classified as one of these four edge types. To to this, we need to keep track of when a vertex is first discovered (visited) in the search (recorded in \(\text{start\_time}[v]\)), and when it is finished (recorded in \(\text{finish\_time}[v]\)), that is, when its adjacency list has been examined completely. These **timestamps** are integers between 1 and \(2|V|\), since there is one discovery event and one finishing event for each of the \(|V|\) vertices.

Tree edges are initially marked when vertices are visited for the first time. Observe that vertices that visited on the forward pass (\(\text{start\_time}\) is marked), but not yet on the backward pass (\(\text{finish\_time}\) is unmarked) form a linear chain of descendants corresponding to the stack of active DFS-\text{VISIT} invocations. Therefore, during execution, if we explore a node that is a descendant, we mark that edge as a back edge. An edge \((u, v)\) is a forward edge, if \(v\) is finished and \(\text{start\_time}[u] < \text{start\_time}[v]\). An edge \((u, v)\) is a cross edge, if \(v\) is finished and \(\text{start\_time}[u] > \text{start\_time}[v]\). The following is the Python code for classifying edges in a directed graph.

```python
class DFSResult:
    def __init__(self):
        self.parent = {}
        self.start_time = {}
        self.finish_time = {}
        self.edges = {}  # Edge classification for directed graph.
```
def dfs(g):
    results = DFSResult()
    for root in g.itervertices():
        if root not in results.parent:
            dfs_visit(g, root, results)
    return results

def dfs_visit(g, v, results, parent = None):
    results.parent[v] = parent
    results.t += 1
    results.start_time[v] = results.t
    if parent:
        results.edges[(parent, v)] = 'tree'
    for n in g.neighbors(v):
        if n not in results.parent: # n is not visited.
            dfs_visit(g, n, results, v)
        elif n not in results.finish_time:
            results.edges[(v, n)] = 'back'
        elif results.start_time[v] < results.start_time[n]:
            results.edges[(v, n)] = 'forward'
        else:
            results.edges[(v, n)] = 'cross'
    results.t += 1
    results.finish_time[v] = results.t
    results.order.append(v)

We can use edge type information to learn some things about $G$. For example, tree edges form directed acyclic graphs or DAGs in which there is exactly one path from the root (line 12) of the trees to every vertex marked in that invocation of dfs_visit (line 14). In addition, $G$ has a cycle if and only if DFS finds at least one back edge.

An undirected graph may entail some ambiguity, since $(u, v)$ and $(v, u)$ are really the same edge. In such a case, we classify the edge as the first type in the classification list that applies, i.e., we classify the edge according to whichever of $(u, v)$ or $(v, u)$ the search encounters first. Note that undirected graphs cannot contain forward edges and cross edges, since in those cases, the edge $(v, u)$ would have already been traversed (classified) during DFS before we reach $u$ and try to visit $v$.

## Topological Sort

Many applications use directed acyclic graphs to indicate precedences among vertices. Often times, we’d like to generate an ordering of vertices such that the parents vertices are guaranteed to come before child vertices. We call such a ordering a topological sort of the vertices. Note that
there may be multiple valid topological sorts for a directed graph. In addition, note that if the graph contains a cycle, then there cannot exist a topological sort.

We can use DFS to generate a topological sort of a graph. Topologically sorted vertices should appear in reverse order of their finishing time. If any back edges are detected, then the graph contains a cycle, so no topological sort exists.

```python
def topological_sort(g):
    dfs_result = dfs(g)
    if 'back' in dfs_results.edges:
        return None
    dfs_result.order.reverse()
    return dfs_result.order
```

This procedure works because line 24 of `dfs_visit` ensures that all children of a vertex are visited and and have smaller `finish_time` than the vertex itself. Conversely, the parents of a vertex must have a larger `finish_time` than the vertex itself. If there is a tree edge from the parent to the vertex, then as noted before, the vertex must have marked in `finish_time` before the parent. If there is not a tree edge from the parent to the vertex, then the parents must have been explored in subsequent `dfs_visit` calls (line 14), and therefore the `finish_time` of the parents must be greater than that of the vertex. By inverting the visitation order, we ensure that all parents are listed before children.