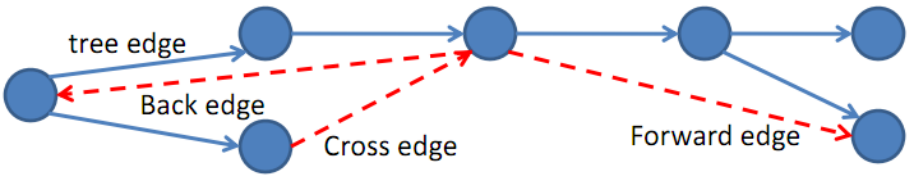
DFS Edge Classification

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. To to this, we need to keep track of when a vertex is first being ***disocvered*** (visited) in the search (recorded in `start_time[v]`), and when it is ***finished*** (recorded in `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 immediate from the specification of the algorithm. For back edges, observe that vertices that are currently being visited but is not finished form a linear chain of descenedants corresponding to the stack of active DFS-VISIT invocations. Exploration always proceeds from the deepest vertex currently being visited, so an edge that reaches another vertex being visited has reached an ancester. An edge (u, v) is a forward edge, if v is finished and $\texttt{start\_time}[u] < \texttt{start\_time}[v]$. An edge (u, v) is a cross edge, if v is finished and $\texttt{start\_time}[u] > \texttt{start\_time}[v]$. The following is the Python code for classifying edges in a directed graph.

```
1 class DFSResult:
2     def __init__(self):
3         self.parent = {}
4         self.start_time = {}
5         self.finish_time = {}
6         self.edges = {} # Edge classification for directed graph.
7         self.order = []
```

```
8           self.t = 0
9
10  def dfs(g):
11      results = DFSResult()
12      for vertex in g.itervertices():
13          if vertex not in results.parent:
14              dfs_visit(g, vertex, results)
15      return results
16
17  def dfs_visit(g, v, results, parent = None):
18      results.parent[v] = parent
19      results.t += 1
20      results.start_time[v] = results.t
21      if parent:
22          results.edges[(parent, v)] = 'tree'
23
24      for n in g.neighbors(v):
25          if n not in results.parent: # n is not visited.
26              dfs_visit(g, n, results, v)
27          elif n not in results.finish_time:
28              results.edges[(v, n)] = 'back'
29          elif results.start_time[v] < results.start_time[n]:
30              results.edges[(v, n)] = 'forward'
31          else:
32              results.edges[(v, n)] = 'cross'
33
34      results.t += 1
35      results.finish_time[v] = results.t
36      results.order.append(v)
```

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**.

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 events. Topologically sorted vertices should appear in reverse order of their finishing time.

```
1   def topological_sort(g):
2       dfs_result = dfs(g)
3       dfs_result.order.reverse()
4       return dfs_result.order
```