Depth First Search (DFS)

Depth first search, like breadth first search, is a graph search algorithm. Unlike BFS, DFS does not return the shortest paths between nodes. However, DFS does have many applications. In lecture, we saw that DFS can be used to detect cycles in a graph, and can also be used to topologically sort nodes in a directed acyclic graph.

Iterative Implementation

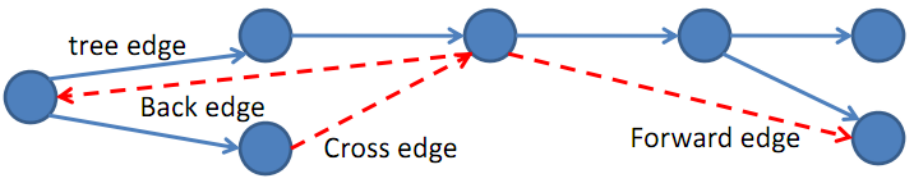
In lecture, we saw an iterative implementation for BFS, and then a recursive implementation for DFS. It turns out that it is possible to implement DFS iteratively, and the code is very similar to the code for BFS. If we take the code from BFS (see recitation 11 notes) and replace the queue with a stack, the result is a correct implementation of DFS-VISIT.

The reason that DFS requires an extra method which iterates through all the nodes and calls DFS-VISIT from each node that has not already been visited is that in DFS we expect to explore the entire graph, even nodes which are not reachable from the start node. If the graph is connected, then DFS will only make a single call to DFS-VISIT.

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 do this, we need to keep track of when a vertex is first being ***discovered*** (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 are not finished form a linear chain of descendants 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 ancestor. 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
```

Let's examine why this algorithm correctly produces a topological sort of the nodes. Consider some edge (u, v) in the graph. We are outputting nodes in their reverse finishing order. For the topological sort to be correct, it must be the case that u comes before v. Thus, it suffices to prove that v finishes before u.

To see why this must be true, let's consider the moment when u has been already started and DFS comes across v. There are three possible scenarios.

1. If v is already finished, then clearly u finishes after v.
2. If v is started but not yet finished, then there must be a path from v to u, which means there is a cycle in the graph, which is a contradiction.
3. If v has not been started, then it will be inserted into the stack after u, and therefore popped from the stack (finishing it) before u.

Thus, we know that u finished after v, and we have a correct topological sort.

Practice Problems

You are given a connected undirected graph $G = (V, E)$ with n vertices and m edges. Give an $O(n+m)$ time algorithm that computes a connected graph $G' = (V, E')$ such that: a) $E' \subseteq E$ and, b) G' has no cycles. That is, your algorithm should remove a subset of the edges of G and make it cycle-free.

Solution: Do a DFS of the graph, and at the end, remove all the back edges. As you traverse the graph, you can check whether the edge you are trying to relax goes to a node that has been seen but is not yet finished, and if so, then it is a back edge and you can store it in a set. After the DFS, remove all the edges that are in this set.

An undirected graph is said to be *Hamiltonian* if it has a cycle containing all the vertices. Any DFS tree on a Hamiltonian graph must have depth $V - 1$.
Solution: False, provable by counterexample

Cycle Testing

Graph G has a cycle iff its DFS has a back-edge.

1. Graph G has a cycle if its DFS has a back-edge. You followed some path from a vertex u to v in the original DFS tree, and now you have a back-edge leading back from v to u–going from a descendant to an ancestor. This is pretty much the definition of a cycle.

2. Graph G's DFS has a back-edge if G has a cycle . If there is a cycle in a graph, then there must be a backwards edge. If there is a cycle that incorporates u and v, then by definition there exists a path from u to v, and from v to u.

Each descendant must be completely explored before it's parent terminates. This argument telescopes.

If u is visited first, u will be a ancestor of v. v must be fully explored before u. After reaching v, we eventually explore the path from v to u. v must terminate first, since it is the descendant of u. The ancestor, u, may not be appended to the stack again, so that an edge to u is classified as a backwards edge.

Checking for Cycles

Checking whether the ancestor node is simply visited is not good enough. The ancestor might be part of a different tree. To check whether the back-edge corresponds to the same connected component, we modify the procedure, so that when visit is called, the node is set to "processing = True" before recursing further. When the visit procedure for a node ends, it is set to "processing = False."