Graph Algorithms, Part I Traversal
Statistics 650/750
Week 4 Thursday

Christopher Genovese and Alex Reinhart

21 Sep 2017

Announcements

- `Stackit!` exercise is available and lots of fun.
- Apologies: My office hours today and Monday are canceled because I am traveling, starting right after class. I will hold some extra hours TBA next week to help make up for the gap.
- Another tooling help session will also be set for next week, time TBA

Brief Review of Last Time

Graphs: Modeling Relationships

Comment on Graph Data Structures

A	B
B	A C U
C	B U E F G
U	B C
E	C F
F	C E G
G	C F

Breadth-First Search (BFS)

We will maintain a **queue** of nodes, initialized with the start node. We successively dequeue a node and process it and enqueue all of that nodes fresh neighbors (processing all the edges along the way).

Inputs:

`graph` an undirected graph

`start` a node at which to start the search

`acc` an accumulator object of arbitrary type

`before_node` a function(node,ts) called when a node is `VISITED`

`after_node` a function(node,ts) called when a node is `PROCESSED`

`on_edge` a function(from,to,ts) called when an edge is traversed ts   a traversal state object (newly initialized if None)

Output:

- An updated traversal state ts'

The algorithm:

```
function bfs(graph, start, acc, before_node=None, after_node=None,
             on_edge=None, ts=None):

  Initialize traversal state ts if not supplied
  Create a new empty queue

  enqueue(start)
  mark start visited
  process start with before_node [if supplied]

  while queue is not empty:
     current_node = dequeue queue

     for neighbors of current_node:
         if neighbor is not processed:
             process edge (current_node, neighbor) with on_edge [if supplied]

         if neighbor is fresh:
             enqueue neighbor
             mark neighbor visited
             set parent[neighbor] = current_node in ts
             process neighbor with before_node [if supplied]

     mark current_node processed
     process current_node with after_node [if supplied]
```

The Traversal Trees

Graph Traversal Continued

Breadth-First Search (cont'd)

The traversal we saw last time will cover a single connected component.

Assume we also have a functions bfs_all that calls bfs on successive fresh nodes, updating the same traversal state and returning it.

```python
def bfs_all(graph, acc, before_node=None, after_node=None, on_edge=None):
    ts = None

    for node in graph.nodes():
        if ts is None or ts.fresh(node):
            ts = graph.bfs(node, acc, before_node, after_node, on_edge, ts)

    return ts
```

What do the parent pointers do here?

```
The node from which node i was visited is assigned to parent[i].
These pointers traces the visitation paths of the algorithm.
```

Examples: How to Use BFS

1. You have a function

   ```python
   def inc(graph, node, ts):
       ts.accumulator += 1
   ```

   and call `tstate = bfs(start, 0, inc)`. What is `tstate.accumulator` after the call?

   When the traversal starts the accumulator is 0. Each time a node is visited, this accumulator is incremented. Hence, this counts the nodes in a connected component of the graph, the one that contains `start`.

2. You have a function

   ```python
   def inc_if_blue(graph, node, ts):
       props = graph.get_node_properties(node)
       if  props['color'] == 'blue':
           ts.accumulator += 1
   ```

   and call `tstate = bfs(start, 0, inc_if_blue)`. What is `tstate.accumulator` after the call?

   As before, we are incrementing the accumulator when we visit nodes, but this time only for blue nodes. Hence, we are counting the number of blue nodes in the connected component containing the node `start`.

3. You have a function

   ```python
   def parents(graph, node, ts):
       parent = ts.parent[node]
       my_name = graph.get_node_properties(node, 'label')
       if parent:
           p_name = graph.get_node_properties(parent, 'label')
       else:
           p_name = None
       ts.accumulator[my_name] = p_name
   ```

   and call `tstate = bfs(start, {}, parents)`. What is `tstate.accumulator` after the call?

   Here, we pass in an empty dictionary and record for each node, its label and the label of its "parent" – the node we came from during BFS. The accumulator thus contains a dictionary mapping node labels to parent labels.

Exercises

1. You have a function

```python
def blue_labels(graph, node, ts):
    props = graph.get_node_properties(node)
    if  props['color'] == 'blue':
        ts.accumulator.append(props['label'])
```

   and call `tstate = bfs(start, [], before_node=blue_labels)`. What is `tstate.accumulator` after the call?

2. Write a function `find_path(start, end, parents)` that takes the BFS tree (through the parent pointers) and returns a list of node IDs giving a path from `start` to `end`, or `None` if there is no such path.

   What kind of path does BFS find?

```python
def find_path(from_node, to_node, parents):
    path = []
    end = to_node
    while from_node != end and end is not None:
        path.append(end)
        end = parents[end]
    if end is not None:
        path.append(from_node)
        path.reverse()
        return path
    else:
        return None

# ts = bfs(...)
# find_path(0, 3, ts.parent)

# Note: This captures the gist, but it fails if from_node is
# not start node for the bfs.

# To make this work for any from_node, notice that three
# things can happen in find_path() above: 1. the path from
# to_node ends at from_node giving a direct path; 2. the
# path from to_node and a similar path from from_node come
# to the same node that is different from either, in which
# case we get a path by joining the two (with one reversal);
# or 3. the paths from the two nodes lead to different
# nodes, in which case, to_node and from_node are in
# different connected components. The modified code
# looks as follows; notice how we refactored the operation
# into two pieces.

def find_bfs_root_path(node, parents):
    """Find path from a node to the root of its BFS tree

       node    -- a node identifier in a graph
       parents -- an array specifying a BFS tree, with
                  parents[nd] the parent of node nd
                  in the tree or None if nd is the root.

       Returns a list of nodes starting with the given node.
    """
    path = []
    end = node
    while end is not None:
        path.append(end)
        end = parents[end]
    return path

def find_path(from_node, to_node, parents):
    to_path = find_bfs_root_path(to_node, parents).reverse()

    if to_path[0] == from_node:
        return path
    else:
        from_path = find_bfs_root_path(from_node, parents)
        if to_path[0] == from_path[-1]:
            return from_path + to_path
        else:
            return None
```

3. Configure BFS to find the **connected components** of a graph, these are the sets of nodes such that within each set there is a path between any two nodes.

```
def collect_visited(graph, node, state):
    """Accumulates list of nodes as they are visited."""
    state.accumulator.append(node)

def grab_component(graph, components, start, state=None):
    """Collect one connected component and reset state accumulator."""

    state = graph.bfs(start, [], before_node=collect_visited, ts=state)
    components.append(state.accumulator)
    state.accumulator = []

    return state

def connected_components(g):
    """Returns a list of connected components for a graph g"""

    components = []
    ts = None
```

```python

    for node in g.nodes():
        if ts.fresh(node):
            ts = grab_component(g, components, node, state=ts)

    return components
```

4. Configure BFS to determine if the graph can be *two-colored*, meaning that we can assign one of two colors to every node without two nodes of the same color sharing an edge between them. A two-colorable graph is said to be **bipartite**. Find the two coloring or return None/null/NA if the graph is not bipartite.

```python
def complementary_color(color):
    return 1 - color

def check_edge(graph, node, neighbor, state):
    node_color = graph.get_node_properties(node, "color")
    nghb_color = graph.get_node_properties(neighbor, "color")

    if node_color == nghb_color:
        ts.accumulator = False  # Bipartite indicator
        ts.finished = True

    graph.update_node_properties(neighbor,
                                 color=complementary_color(node_color))

def two_coloring(g):
    """Returns a two-coloring of a graph g if bipartite, else False."""

    ts = None

    for node in g.nodes():
        if ts is None or ts.fresh(node):
            g.update_node_properties(node, color=0)
            ts = g.bfs(node, True, on_edge=check_edge, ts=ts)

        if ts.finished:
            break

    if ts.accumulator:
        return [(node, g.get_node_properties(node, "color")) for node in g.nodes()]
    else:
        return False
```

Depth First Search (DFS)

In contrast to BFS, in DFS we will maintain a **stack** of nodes, initialized with the start node.

We successively pop a node and process it and push all of that nodes fresh neighbors (processing all the edges along the way). There is a recursive logic to DFS: for each fresh neighbor, call DFS on it (maintaining state).

```
DFS(start):
  for neighbor in neighbors(start):
     if neighbor is FRESH:
        DFS(neighbor)
```

Wait, where's the stack?

For our algorithm, we take the inputs:

`graph` an undirected graph

`start` a node at which to start the search

`acc` an accumulator object of arbitrary type

`before_node` a function(node,ts) called when a node is VISITED

`after_node` a function(node,ts) called when a node is PROCESSED

`on_edge` a function(from,to,ts) called when an edge is traversed

`ts` a traversal state object (newly initialized if None)

We output an updated traversal state `ts'`.

```
function dfs(graph, start, acc, before_node=None, after_node=None,
             on_edge=None, ts=None):

  tick the clock
  state[node] = VISITED
  visited_time[node] = time

  do before_node processing of node [if supplied]

  for each neighbor of node:
      do on_edge processing of edge(node <-> neighbor) [if supplied]

      if state[neighbor] is FRESH:
          parent[neighbor] = node
          dfs(graph, neighbor, acc, before_node, after_node, on_edge, ts)

  state[node] = PROCESSED
  tick the clock
  processed_time[node] = time

  do after_node processing of node [if supplied]
```

Alternately, we can explicitly use a stack, looping until the stack is empty:

```
function dfs(graph, start, acc, before_node=None, after_node=None,
             on_edge=None, ts=None):
  time = 0
  stack is empty
```

```
if ts is None initialize traversal state:
    state of all nodes = FRESH
    parent[start] = None
    accumulator = acc
    finished = False
else:
    use ts as traversal state

push (start, True) onto stack

while stack is not empty and not finished:
    peek at (current, is_node?) on top of stack

    if is_node? is False:
        do on_edge processing of current edge (if specified)
        pop the stack
    else if state[current] is FRESH:
        tick the clock
        state[current] = VISITED

        do before_node processing of current(if specified)

        for each neighbor of current:
            if neighbor is FRESH:
                parent[neighbor] = current
                push (neighbor, True) on stack
                push (edge[current<->neighbor], False) on stack

    else if state[current] is VISITED:
        tick the clock
        state[current] = PROCESSED
        do after_node processing of current (if specified)
        pop the stack
    else:
        pop the stack

return traversal state
```

Again, suppose we have `dfs_all` which continues searching until no fresh nodes are found:

```python
def dfs_all(self, acc, before_node, after_node, on_edge):
    ts = None

    for node in self.nodes():
        if ts is None or ts.fresh(node):
            ts = self.dfs(node, acc, before_node, after_node, on_edge, ts)

    return ts
```

Example: How to Use DFS

1. Task: Print traversal history as DFS runs

   Basic idea: Mark each node as it is being visited and processed, and mark each edge as it is being traversed. Here, we will use node labels to keep track.

   Solution: See print-history.py for the solution.

2. Task: Detect cycles in a graph with DFS.

   Basic Idea: In an undirected graph, look for edges that creates a cycle.

```python
def detect_cycle_edge(g, from_node, to_node, ts):
    if ts.visited(to_node) and ts.parent[from_node] != to_node:
        from_label = gr.get_node_properties(from_node, 'label')
        to_label = gr.get_node_properties(to_node, 'label')

        print("Found cycle with nodes {from_n}"
              "and {to_n}".format(from_n=from_label, to_n=to_label))

        ts.finished = True
```

   Pass this as the `on_edge` argument.

Exercise: Directed Graphs

How would we change the DFS algorithm above for use with digraphs?

Answer: The algorithm itself does not change, but we can no longer guarantee that the entire connected component will be reached from any given source node. See the discussion of Strongly Connected Components below.

DAGs and Topological Sort

A **topological sort** of a DAG is a linear ordering of the DAG's nodes such that if (u, v) is a directed edge in the graph, node u comes before node v in the ordering.

Example

Given a DAG, how do we use DFS to do a topological sort?

Algorithm `topological-sort`:

```
Input: A DAG G
Output: A list of nodes representing a topological sort

Steps: Run DFS on G, configured with after_node so that
after each node is processed, we push it onto the front
of a linked list (or equivalently onto a stack).

Return the list of nodes.

(Note: We can also use the timestamps, specifically the
 processed_times to get a valid sorting.)
```

Exercise: Code or Pseudo-Code this after-node function

Other Applications and Exercises

1. Configure `dfs` to count the number of "descendants" of a node.
2. Configure `dfs` to compute a path between two nodes. What kind of paths does DFS produce?

Application: Directed Graphs and Strongly Connected Components

A directed graph is **weakly connected** if the corresponding *undirected* graph (replacing directed edges with undirected edges) is connected.

A directed graph is **strongly connected** if for any two nodes v and w there is a *directed* path from v to w and from w to v.

The strongly connected components of a directed graph are its maximal, strongly connected subgraphs. We can find the strongly connected components with two DFS's.

Let G be a directed graph and let G' have the same nodes and edges with all the edges *reversed.* Pick an arbitrary node v.

The algorithm for *detecting* strong connectivity is basically as follows:

1. Do `DFS(G, v)`.
2. If the traversal does not contain all nodes, then there are nodes we cannot reach from v. Hence, G cannot be strongly connected.
3. Do `DFS(G', v)`.
4. If this traversal does not contain all nodes, then there are nodes in G from which we cannot reach v. Hence, G cannot be strongly connected.

To find the strongly connected components, we just do a little processing.

In step 1, record the `processed_time`'s. In step 3, do `DFS_ALL(G')` with the nodes ordered as the reversal of the `processed_time`'s.

Other Traversal Schemes

Stacks and queues impose an ordering on how we take data out: LIFO and FIFO respectively. We can view these as **priority queues** that assign a score to every item put in them and extract an item with the highest (or wlog lowest) score.

Question: How are priorities assigned for stacks and queues?

We can thus see BFS and DFS as part of a continuum. If we maintain a general priority queue of prospecive nodes, we get a wide variety of different traversal schemes.

The *same basic algorithm* can be used with just minor modifications.

Group Discussion: How would we change the BFS algorithm to do a Priority First Traversal?

Minimum Spanning Tree (for Weighted Graphs)

A *spanning tree* of a connected, undirected graph G is a subgraph of G that is a *tree* connecting every node. (A general undirected graph thus has spanning *forests.*)

If the edges of G are weighted, then a **minimum spanning tree (MST)** is a spanning tree with minimal *sum* of edge weights.

In general, there can be more than one MST for a graph G, but under some conditions (e.g., distinct edge weights), one can show uniqueness.

Questions:

1. Every connected graph has a spanning tree. Why?
2. Conceptually, how might we go about finding an MST?

Prim's Algorithm as Priority Queue Traversal

We build a tree from an arbitrary root by successively adding the shortest edge leaving the tree.

We use a priority queue to efficiently find which edge to add to the tree. Only nodes that are not in the tree are contained in the priority queue. We build the tree through a `parents` table, as in the traversal algorithms, and we maintain a table `priority` that is the minimum edge weight connecting each node (in the queue) to the tree.

```
Inputs: G     a (weighted, undirected) graph
        root  a node in G
Output: A parents list specifying a minimum spanning tree
        from root, with parents[w] = v if the edge from
        v to w was added to tree.

# Initialize priority queue
Q = new priority queue
foreach node n in G:
    if node is root:
        priority[node] = 0
    else:
        priority[node] = Infinity
    parents[node] = nil
    Q.add(node, priority[node])

# Build the tree
while Q is not empty:
    closest = Q.extract_minimum()
    foreach neighbor n of closest:
        if n is in Q and G.weight(closest,n) < priority[n]:
            parents[n] = closest
            priority[n] = G.weight(closest,n)
```

Classic Algorithms: Prim and Kruskal

Prim's algorithm builds a tree one node at a time. An alternative is Kruskal's algorithm which starts with a forest containing each node, and merges trees together. The classical implementations have a trade off, making one better for sparse and one for dense trees. A good priority queue implementation of Prim's however, manages to be a good default algorithm in both cases.

Prim's Algorithm:

```
Pick a node to be the root of the tree
While the tree does not contain every vertex:
  Find the shortest edge leaving the tree
  Add it to the tree
```

Running time is $O((|N| + |E|) \log |N|)$.

Kruskal's Algorithm:

```
Create a forest F consisting of each node in G as a separate tree
Create a set S of every edge in G
while S is not empty and F is not complete (i.e., spanning):
  Remove edge e with minimum weight from S
  Add e to the forest
  If the removed edge connects different trees in the forest:
    combine the trees
```

Single-Source Shortest Paths

Given a weighted, directed graph G and a specific node s, we want to find the shortest path from s to each other node in the graph. We do the same basic algorithm as Prim's priority queue version for MST, except instead of a tree, we maintain a path.

```
Inputs: G       a (weighted, directed) graph
        source  a node in G
Output: A list predecessor specifying the shortest paths
        from source, with predecessor[w] = v if we added
        an edge from v to w to the path.

# Initialize priority queue
Q = new priority queue
foreach node n in G:
    if node is source:
        distance[node] = 0
    else:
        distance[node] = Infinity
    predecessor[node] = nil
    Q.add(node, distance[node])

# Build the path
while Q is not empty:
    closest = Q.extract_minimum()
    foreach neighbor n of closest:
        if n is in Q:
            est_dist = distance[closest] + G.weight(closest,n)
            if est_dist < distance[n]:
                distance[n] = est_dist
                predecessor[n] = closest
                Q.decrease_priority(n, distance[n])
```

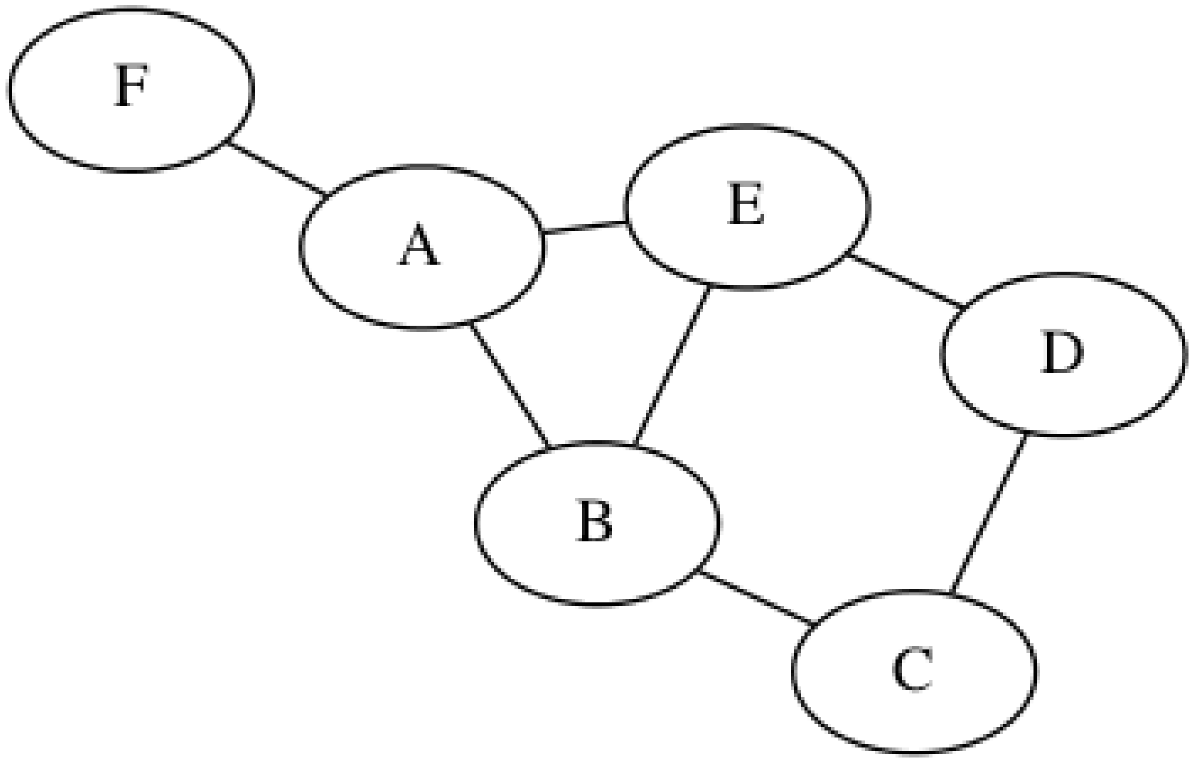
Appendix: Classifying Tree Edges

The BFS Tree (forest)

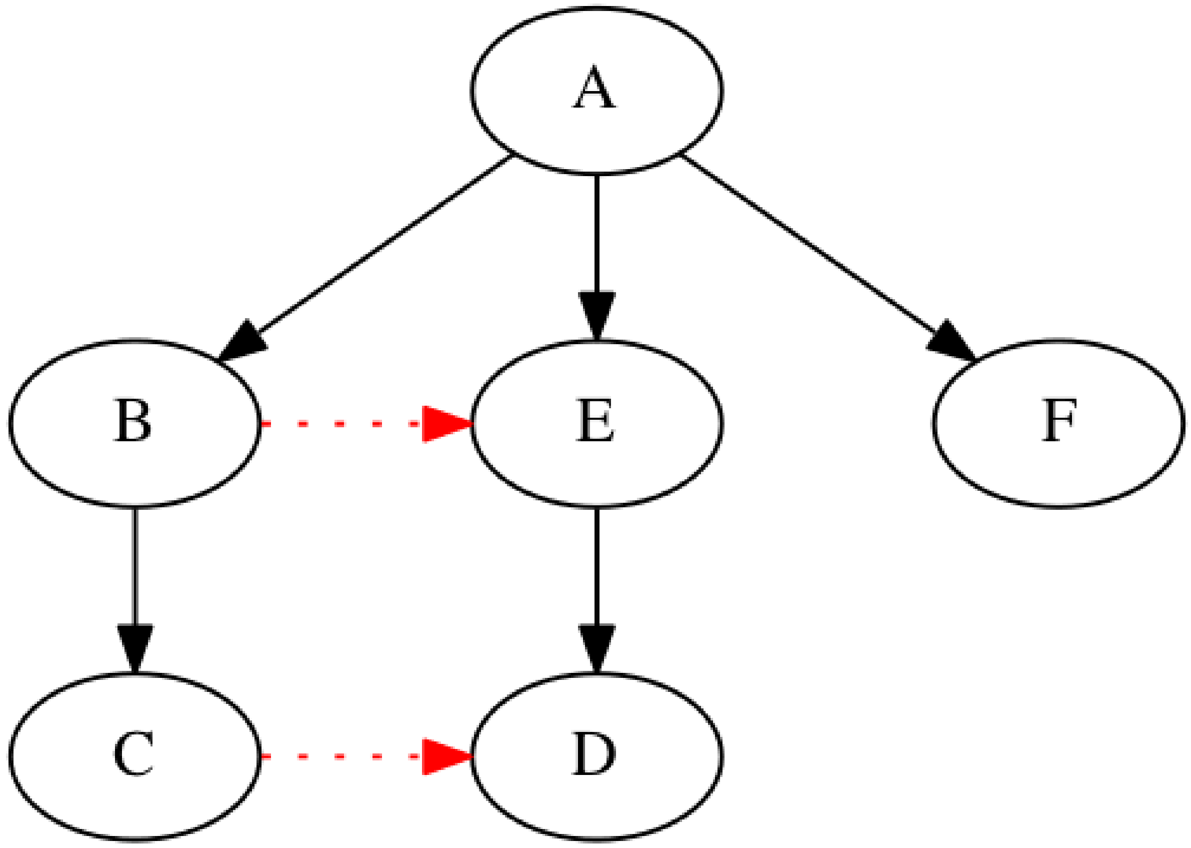
Every node – except the starting node – has a non-null parent. The subgraph consisting of all nodes and "parent" edges (from traversing with `bfs_all`) is thus acyclic and has one fewer edges than each connected component. The components of this subgraph thus form a *tree*, and the whole subgraph is a *forest*. This is called the **BFS tree (forest)** and it has a useful property.

Within any component, the unique path between a node and the starting node (in that component) uses the smallest number of edges of any path between those nodes.

What does the BFS tree look like for this graph?



Here is the BFS tree starting at node A:



Tree edges are solid black, and cross edges are dotted red. We will see a complete edge classification below.

The DFS tree (forest)

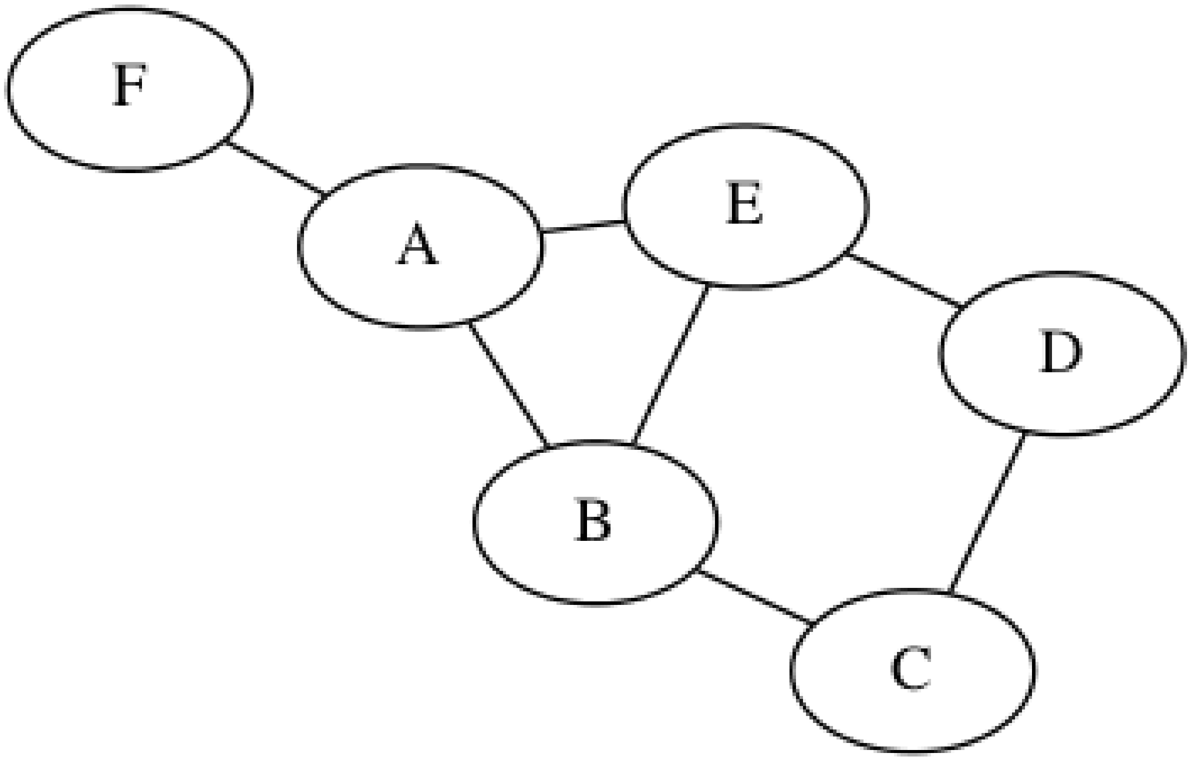
DFS partitions all edges in an undirected graph into two types:

- tree edges – those in the search tree parent structure
- back edges – those not, edges from a node to its ancestor in the tree

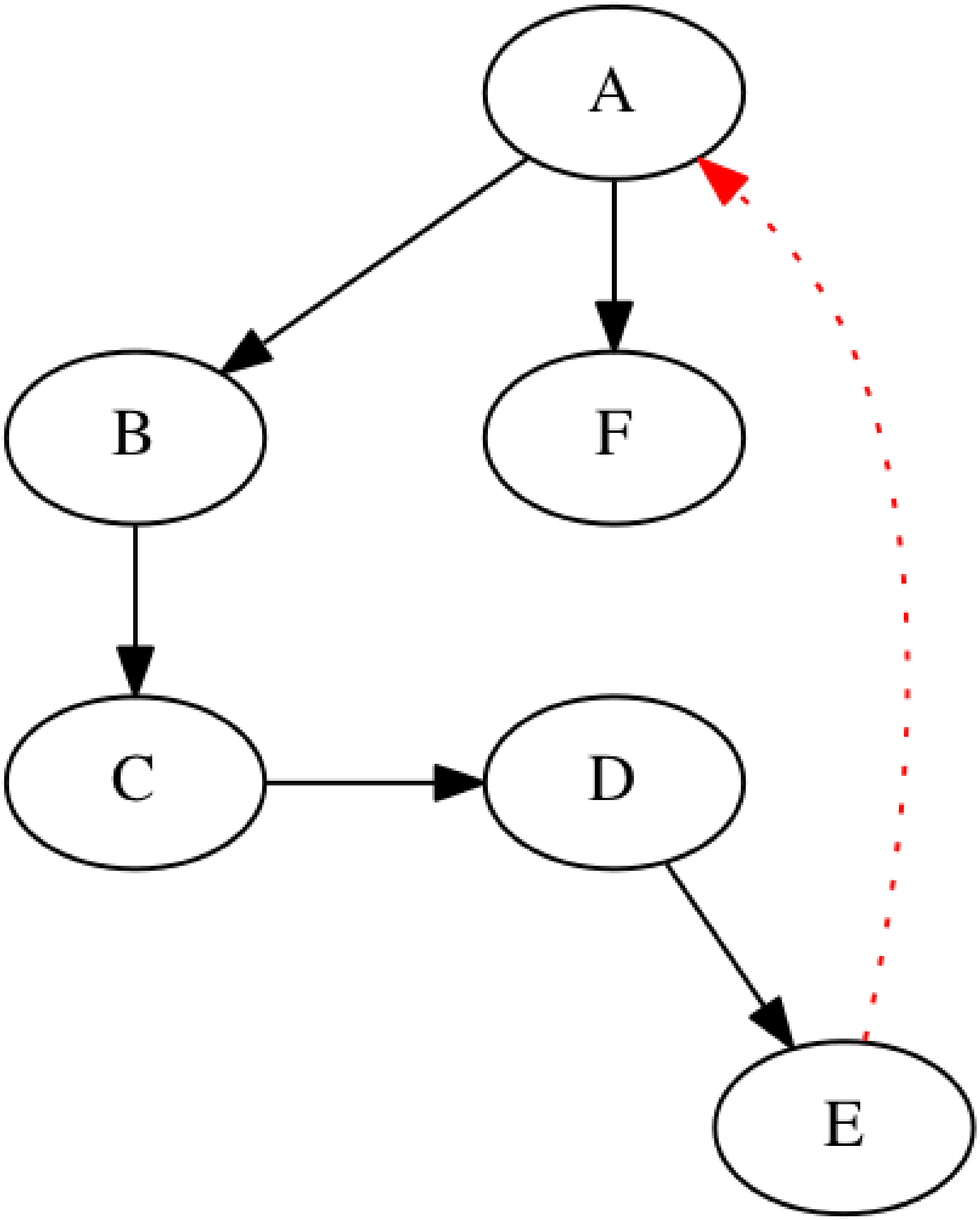
This partition is a useful feature of DFS. Again, we get a tree (forest) from the tree edges.
There are two other kinds of edges that can appear in other cases:

- forward edges – non-tree edges connecting a node to its descendant in the DFS tree
- cross edges – all other edges

Question: What does the DFS tree look like for this graph?



Here's the DFS tree starting at A:



Tree edges are solid black, and back edges are dotted red.
Question: In terms of the states and parent structure, how do we detect a back edge between two nodes?
Question: What do the “clock” times mean? What use are they?

Connectivity

The **connectivity** of a graph is the smallest number of nodes that must be removed (along with their incident edges) so that the graph is no longer connected.

If a graph has connectivity 1, then there is at least one node – called an *articulation node* or *cut node* – whose removal will break the graph in two.

How can the DFS tree help us determine if a graph has cut nodes?

Special Case: All edges are tree edges. What does this tell us?

General Case. In general, we have to consider three types of nodes:

- Root cut node: If the root of the DFS tree has two or more children, it is a cut-node.
- Parent cut node: If the earliest reachable ancestor of node v (by tree and back edges) is the parent of v, the the parent must be a cut-node.
- Bridge cut node: If the earliest reachable ancestor of node v (by tree and back edges) is v itself, then parent[v] must be a cut-node.

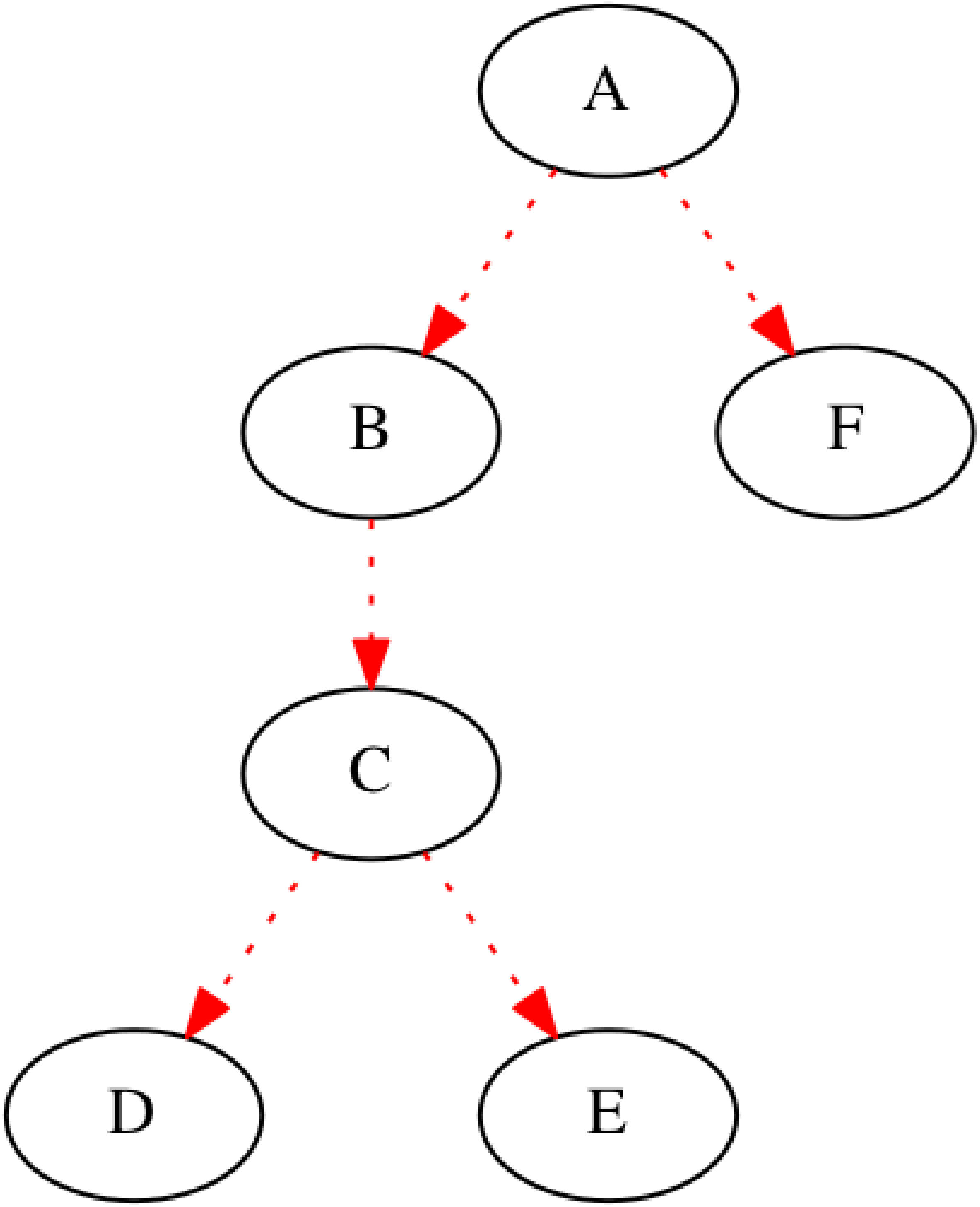
We can encode this with `before_node`, `after_node`, and `on_edge` by keeping track of each nodes earliest reachable ancestor (using *directed* DFS tree and back edges) and the "out degree," the number of directed tree and back edges that leave the node.

```python
# before_node
def init_ancestors(graph, node, state):
    state.reachable_ancestor[node] = node

# on_edge
def update_ancestors_and_degree(graph, from_node, to_node, state):
    edge_type = graph.edge_classification(from_node, to_node) # cf. later

    if edge_type == TREE:
        state.out_degree[from_node] += 1

    if edge_type == BACK and state.parent[from_node] != to_node:
        if state.visited_time[to_node] < state.visited_time[state.reachable_ancestor[from_node]]:
            reachable_ancestor[from_node] = to_node

# after_node
def check_cuts(graph, node, state):
    if state.parent[node] is None:  # root of DFS tree
        if state.out_degree[node] > 1:
            state.accumulator.append((node, "root"))  # found a cut-node
        return

    if state.reachable_ancestor[node] == state.parent[node] and parent[parent[node]] is not None:
        state.accumulator.append((node, "parent"))

    if state.reachable_ancestor[node] == node:
        state.accumulator.append((parent[node], "bridge"))
        if state.out_degree[node] > 0: # not a leaf
            state.accumulator.append((node, "bridge"))

    time_node   = state.visited_time[state.reachable_ancestor[node]]
    time_parent = state.visited_time[state.reachable_ancestor[parent[node]]]

    if time_node < time_parent:
        reachable_ancestor[parent[node]] = reachable_ancestor[node]

# set initial accumulator to []
```

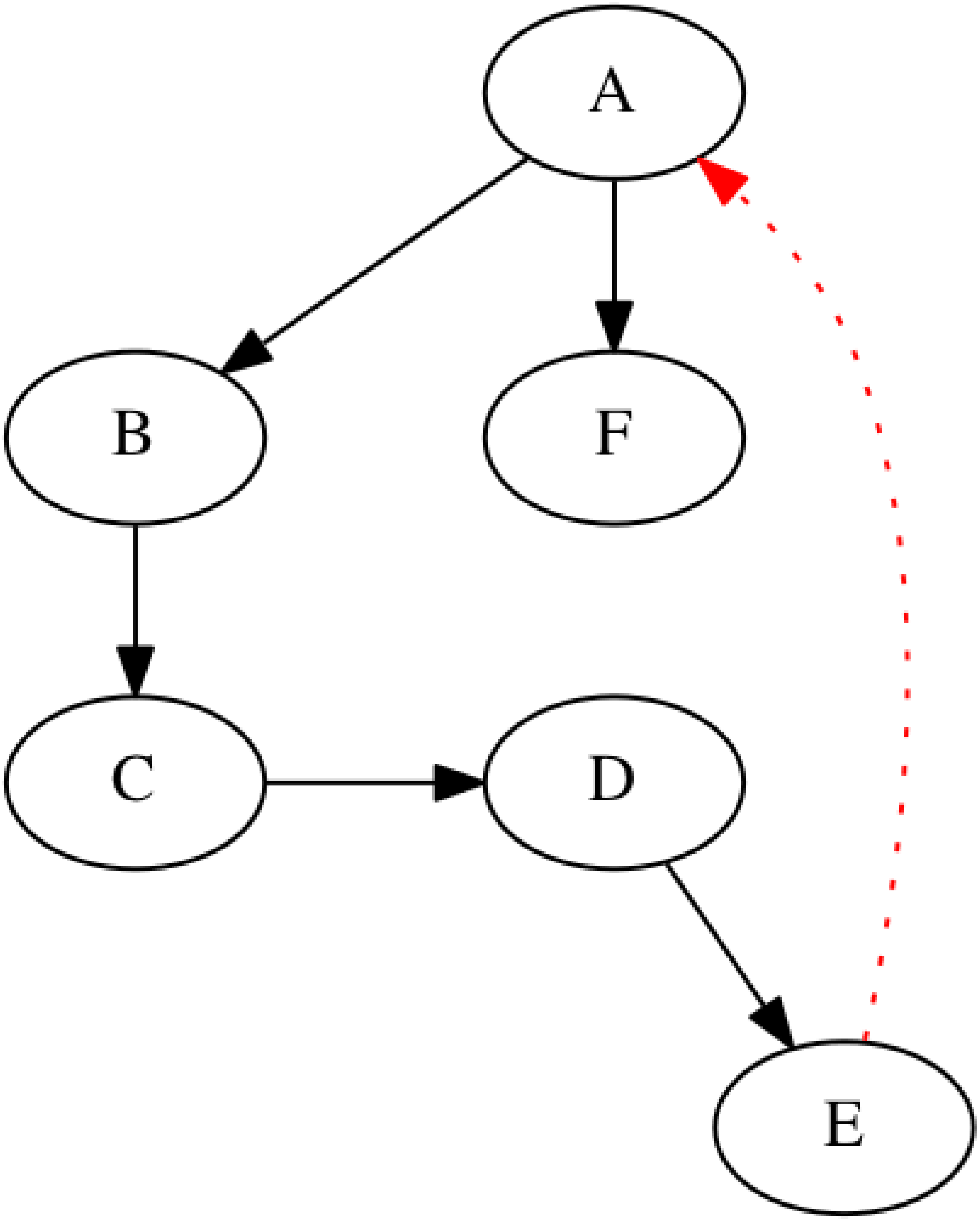
Edge Classification for Directed Graphs

Edges of each type shown in dotted red.

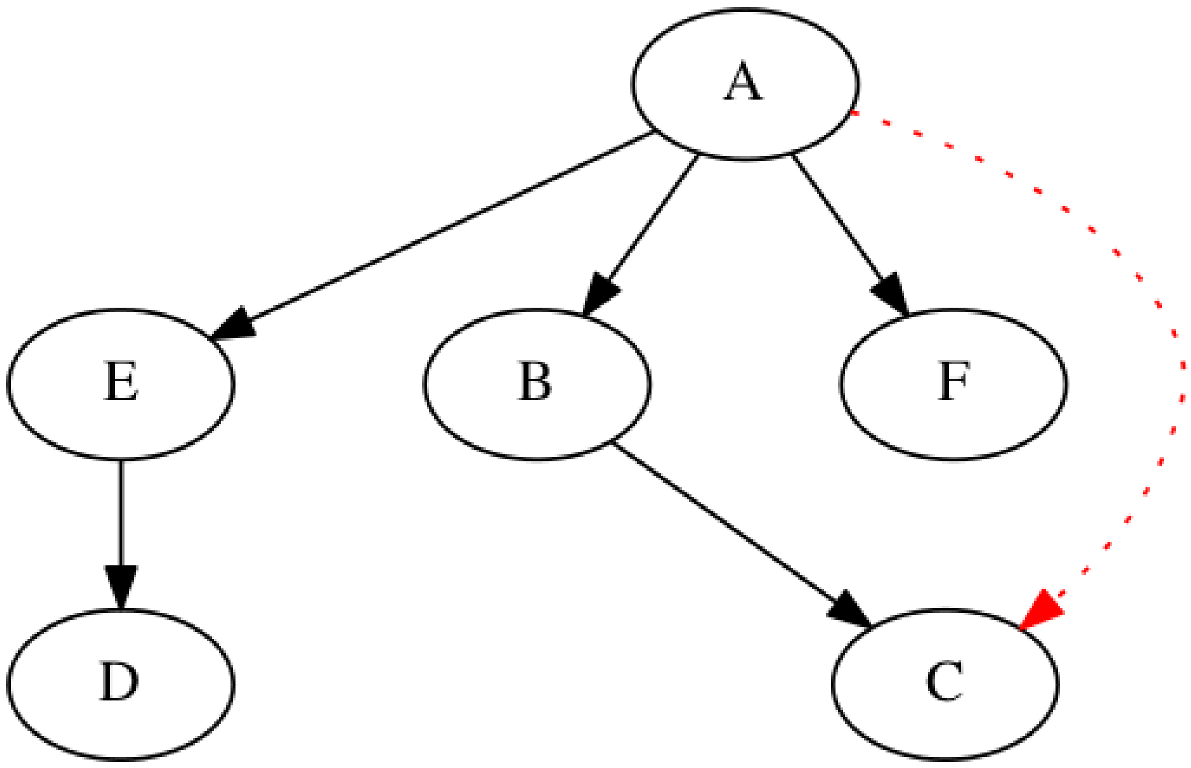
Tree Edges



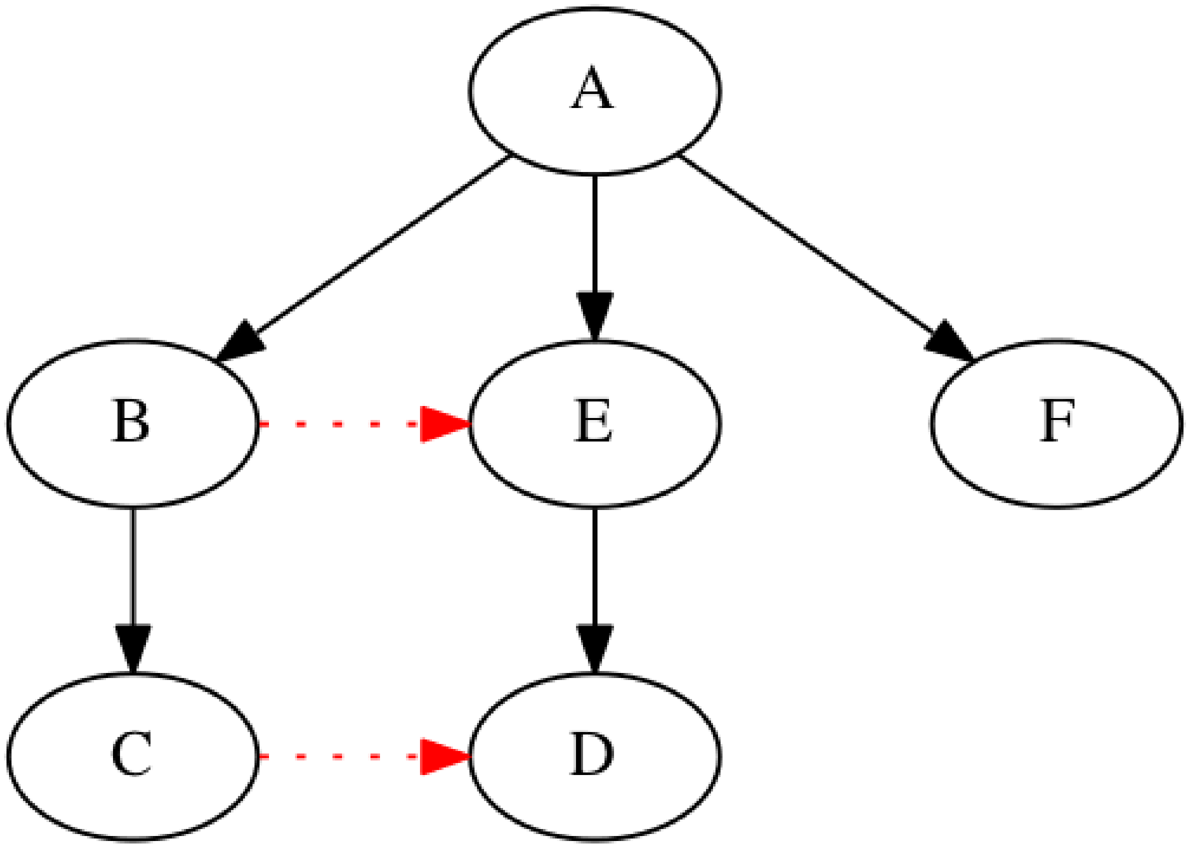
Back Edges:



Forward Edges:



Cross Edges:



Question: For an undirected graph, why are all edges tree or back edges? Why not forward edges or cross edges?

```
Suppose we encounter a ``forward edge'' (u,v), where
v is a descendant of u in the tree. We would have
explored that edge when we visited v making it a back edge.

Suppose we encounter a ``cross edge'' (u,v) linking
unrelated nodes. Then we would have explored that edge
when visiting v, making it a tree edge.
```

Appendix: More on Trees

Trees are probably the most important non-sequential data structure in the study of algorithms. Trees are composed of nodes and edges, and are a special case of Graphs, which we will study in the next couple weeks. But the constraints/structure that define trees arise frequently and thus make them worth studying on their own.

Here, I want to consider several different, though related, definitions of trees that are useful in subtly different circumstances. There is a variety of nomenclature here, covering a range of similar ideas. Loosely speaking, the main divide in the taxonomy is between trees used for *data structures and algorithms*, which have somewhat more specialized definitions, and trees used in *graph theory* which are more general but also more abstract.

For data structures and algorithms, we have a general notion of a *tree* as a hierarchical data structure. We also have a specific definition of *binary tree*. Somewhat confusingly, a *binary tree* is not (quite) a *tree*, for a simple reason to be seen below.

In graph theory, a tree is a connected, acyclic, simple graph. But we can vary a number of properties – labeled vs. unlabelled, rooted vs. unrooted, directed vs. undirected – that lead to special cases.

Trees as Hierarchical Data Structures

Trees and Forests

A **tree** T is a finite, nonempty set of nodes such that

1. One specially designated node is called the *root* of the tree, root(T).
2. The remaining nodes (excluding root(T)) are partitioned into $S \geq 0$ disjoint sets $T_1, ..., T_S$, each of which is a tree.

The trees $T_1, ..., T_S$ are called the **subtrees** of the root. The roots of these trees are called the **children** of the root, and in turn the root is their **parent**. Nodes with no subtrees are called **leaf** (sometimes terminal) nodes; other nodes are called **branch** (sometimes non-terminal) nodes.

If the subtrees of nodes are considered in a specific order, the tree is called *ordered.* This is the most common case, in which two trees with permuted subtrees are considered different.

The **degree** of a node is the number of subtrees it has. The level of a node is defined recursively with root(T) at level 0 and the subtrees of a node N at level level$(N) + 1$.

A **forest** is a set (usually ordered) of zero or more disjoint trees. Note that a tree, as defined above, cannot be empty but that a forest can be. A tree with its root removed creates a forest, and joining the trees in a forest as a subtrees of a new node creates a tree.

Binary Trees

A **binary tree** B is a finite set of nodes that is either empty or consists of a root and the elements of two disjoint binary trees called the left and right sub-trees.

This recursive definition is worth careful consideration. Notice, for instance, that the binary trees

```
  A                    A
 /                      \
B                        B
```

are different binary trees but as trees above, they would be equivalent. The other difference with trees defined above is that binary trees can be empty but trees cannot. The latter condition allows a well-constructed definition of a forest; the former allows for empty sub-trees on the left or right of a binary tree branch.

- Binary tree traversal

  A common operation given a tree is to "visit" all the nodes in the tree (presumably doing something with the information stored in those nodes). This operation is called *traversing* the tree.

  There are many different orders in which one can traverse a tree, but three are particularly valuable in practice:

  - Preorder: Visit Root, Visit Left Subtree, Visit Right Subtree
  - Inorder: Visit Left Subtree, Visit Root, Visit Right Subtree
  - Postorder: Visit Left Subtree, Visit Right Subtree, Visit Root

  As you can see, the name refers to when the root is visited relative to its subtree.

Trees in Graph Theory

A **tree** is a kind of graph with a special structure. The nodes in that graph may be labeled with arbitrary information or may be entirely unlabeled. Here, we are only consider so-called simple graphs that have at most one edge between any pair of nodes and no edges from a node to itself.

We will distinguish between two kinds of trees:

- Unrooted (or free) trees
- Rooted (or oriented) trees.

Unrooted (or Free) Trees

An **unrooted** tree is a connected, acyclic graph T. This is equivalent to:

- T is a minimal connected graph (that is, it is connected but if any edge were removed, the graph would no longer be connected).
- For any two distinct nodes $v \neq v'$, there is exactly one path in T with no repeated nodes between v and v'.

And if T has a finite number $n > 0$ of edges, then these are equivalent to:

- T is acyclic and has $n - 1$ edges
- T is connected and has $n - 1$ edges.

As the name suggests, in an unrooted tree T, we do not distinguish any particular node as the root.

Rooted (or Oriented) Trees

A **rooted** tree is a directed graph T with a designated node r, called the *root*, such that:

- Each node $v \neq r$ is the starting node of exactly one directed edge.
- r is not the starting node of any edge.

It follows that for every node $v \neq r$, there is a unique directed path from v to r.

Given an unrooted tree, we can designate any one node as an the root and assign directions to the edges in a unique way to create a rooted tree. Conversely, we can consider a rooted tree as an undirected graph, which is an unrooted tree.