COMPSCI 250: Introduction to Computation

Lecture #25: DFS and BFS on Graphs

David Mix Barrington and Ghazaleh Parvini

1 November 2023

DFS and BFS on Graphs

- (last four slides of Lecture #24)
- Storing the Entire Search Space
- The DFS Tree of a Undirected Graph
- The DFS Tree of a Directed Graph
- Four Kinds of Edges
- The BFS Tree of a Undirected Graph
- The BFS Tree of a Directed Graph

Breadth-First Search

- Once we reach the distance of the nearest goal node, we will look at *all* nodes at that distance and thus find that goal node.
- Thus we find the *shortest* path, in terms of number of edges.
- But if different edges have different costs, this may not be the *cheapest* path.

Comparing DFS and BFS

- Depth-first search might be much faster if its greedy search succeeds immediately -- breadth-first search *must* check all paths shorter than the right one.

- BFS also uses much more memory in general, as all the nodes at a given distance are stored on the queue at once.

- Without recognizing already-seen nodes, BFS and DFS take about the same time on our example. This is because they put a node on the open list once for each path to it.

Iterative Deepening DFS

- When we can't recognize already-seen nodes, a hybrid approach between DFS and BFS, called **iterative deepening DFS**, can combine the advantages of both.
- The idea is to carry out a DFS but **truncate** it at distance 1. If that fails, DFS again truncating to distance 2, then distance 3, and so on. Like BFS, this is guaranteed to find a shortest path in terms of number of edges.

Iterative Deepening DFS

- We only need to keep a stack rather than a queue. If the graph has degree d, the stack for the distance-k DFS will have at most k nodes on it, while the queue for the corresponding BFS might have as many as d^n nodes on it.
- We appear to be wasting time by doing all the shorter searches before we discover the right distance. But since these searches get exponentially longer with k, the distance-k one takes more time than all the others put together. So we waste only a small fraction of the time for the right search.

Storing the Entire Search Space

- In COMPSCI 311 you'll spend considerable time on search problems where the entire graph is given to you, usually as an **adjacency list** where for each node we have a list of the edges out of it.

- Given two nodes s and t in the graph, we can ask whether there is a path from s to t, how long the shortest path from s to t might be (measured by number of edges or measured by the total cost of the edges), or whether s and t remain connected if certain edges are deleted.

Storing the Entire Search Space

- With the whole graph stored (or using a **closed list** to remember what we've seen), we avoid processing the same node twice.
- Both DFS and BFS on graphs will allow us to create a **tree** from the graph, which will allow us to address these various problems more easily.

DFS Trees of Undirected Graphs

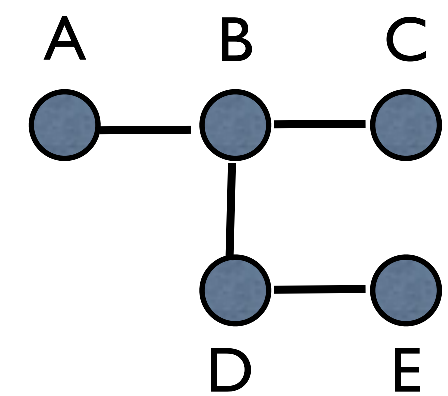
- Recall that our DFS algorithm places nodes onto a stack when they are discovered, and processes all their edges when they are taken off the stack.
- Our DFS tree will have a **tree edge** from s to t if we encounter t for the first time while we are processing s, that is, if we discover t through its edge from s. The tree edges form a tree that gives a path from the start node to each node that is reachable from it.

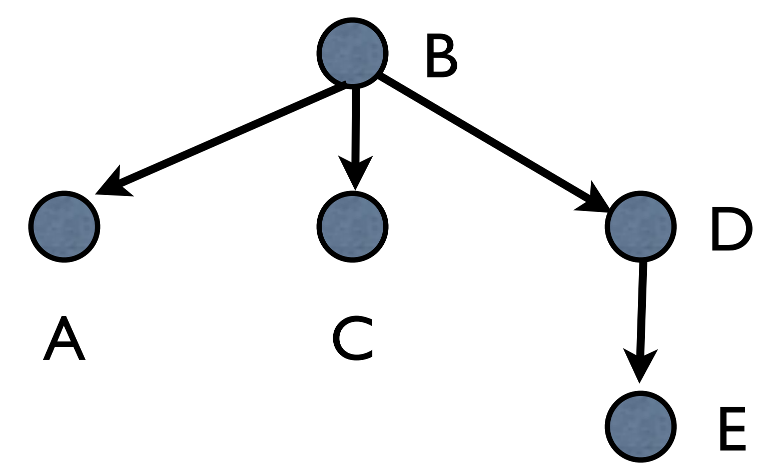
DFS Trees of Undirected Graphs

- If we defined the DFS recursively, the DFS tree would be essentially the call tree, because if (s, t) were a tree edge we would make the recursive call with parameter t in the course of processing the call with parameter s.
- A DFS of an undirected graph searches the entire **connected component** of the start node. What can we tell about the edges that aren't tree edges?

Tree Edges and Back Edges

- Let G be a connected undirected graph and let T be its DFS tree.
- If G were a graph-theoretic tree, T and G would be the same graph (more precisely, T would be the rooted tree made from G with the start node as root).



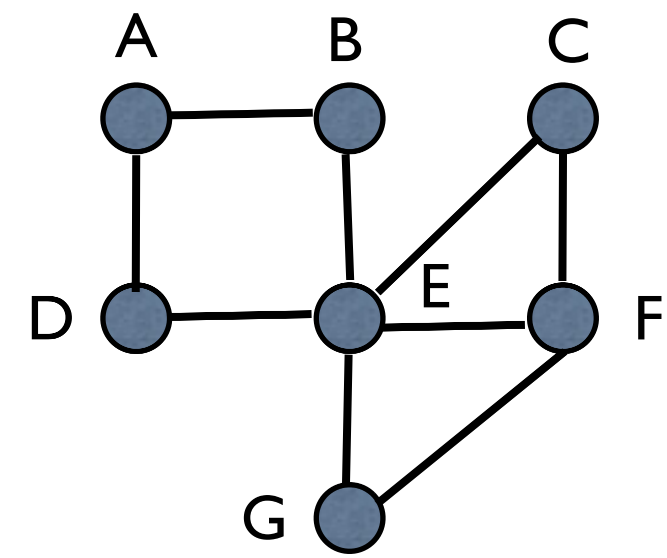


Tree Edges and Back Edges

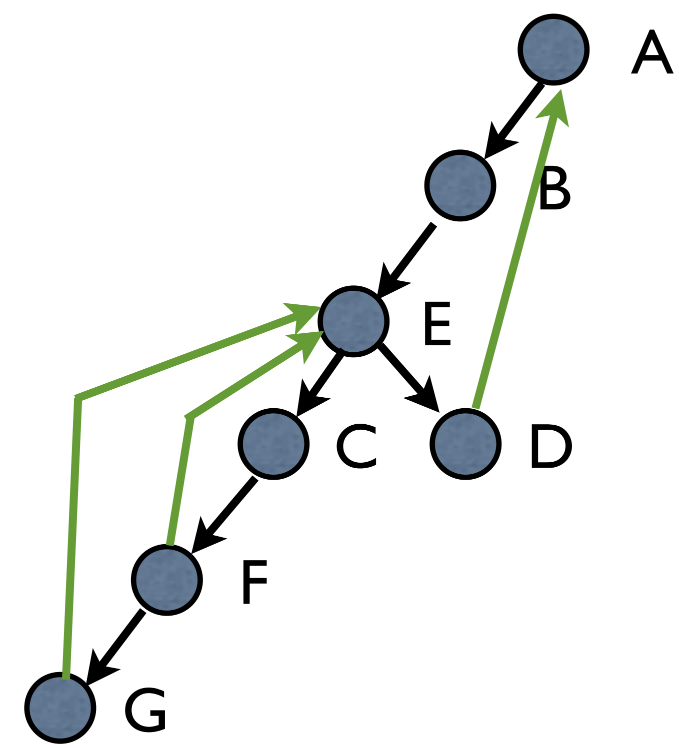
- But if while processing node s, we find an edge to a node t that is not new, that edge does *not* go into T. (We'll ignore the reverse directions of tree edges.)
- Note that the processing of t must still be going on at this point, because we don't finish processing t until we've finished all the nodes reachable from it, including s. So t must be an **ancestor** of s in the tree, and the edge (s, t) is thus called a **back edge**.

Tree Edges and Back Edges

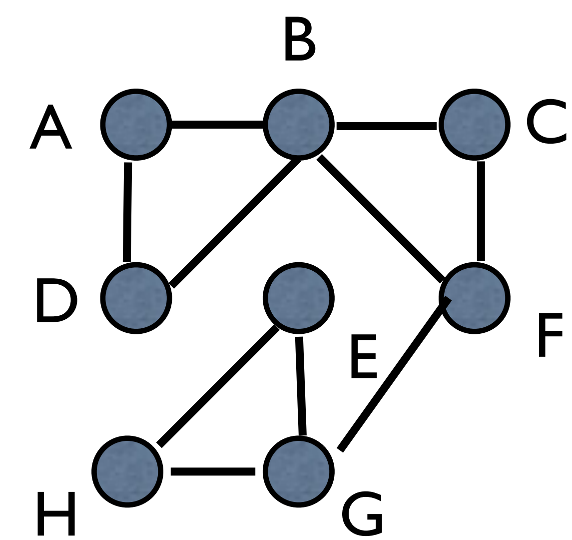
- Here's an example where the undirected graph G becomes a rooted tree T together with some back edges.



- An **articulation point** is a node whose removal disconnects the graph. Can you tell what condition on the tree and back edges makes a node such a point?



Clicker Question #1

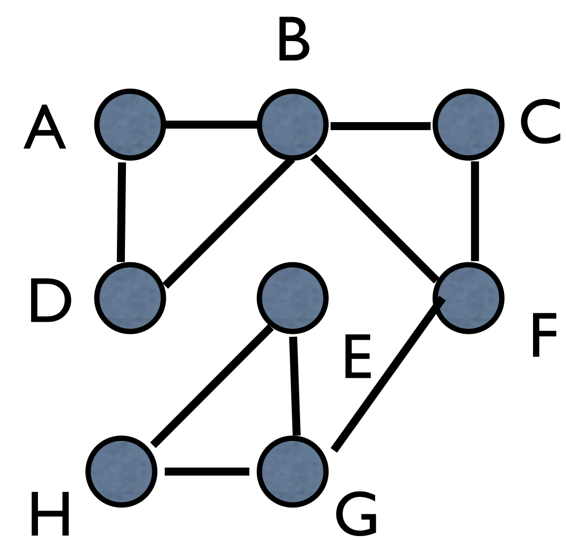


- Let G1 be the graph on the right and let G2 be G1 with added edge B-E. What nodes are articulation points of G1? What about G2?
- (a) G1: B, F   G2: none
- (b) G1: B, F   G2: F
- (c) G1: B, F, G G2: none
- (d) G1: B, F, G G2: B

Not the Answer

Clicker Answer #1

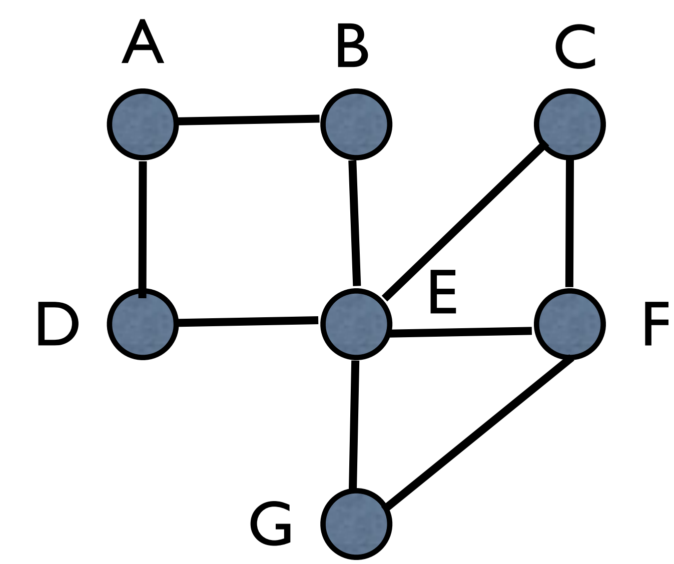
- Let G1 be the graph on the right and let G2 be G1 with added edge B-E. What nodes are articulation points of G1? What about G2?

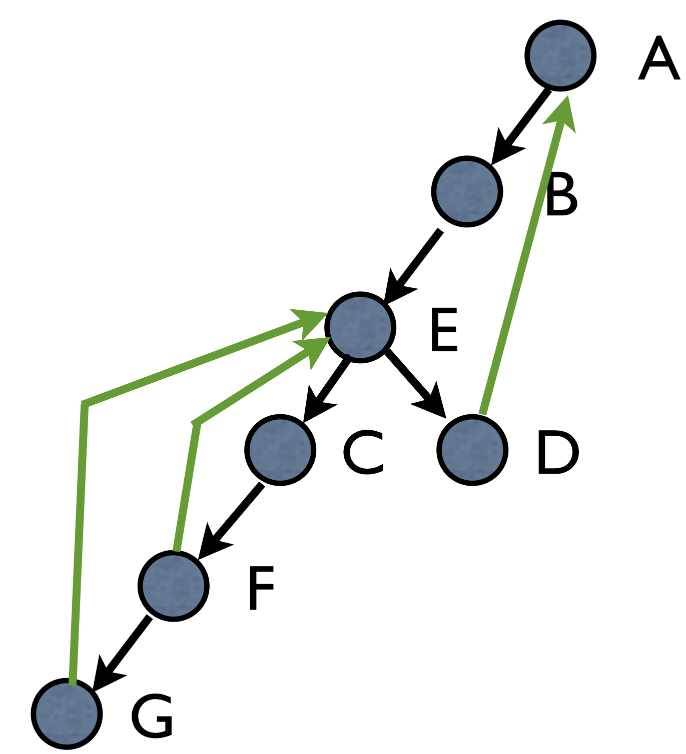


- (a) G1: B, F G2: none deleting still isolates E, H
- (b) G1: B, F G2: F new B-E edge bypasses F
- (c) G1: B, F, G G2: none deleting B still isolates A, D
- (d) G1: B, F, G G2: B

DFS and Articulation Points

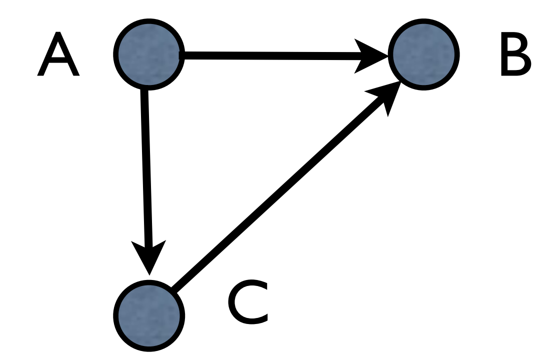
- In this graph, E is the only articulation point.
- Every other node X in the DFS tree (except the root A) has this property: Every child of X has a descendant with a back edge to a proper ancestor of X.
- The root is an articulation point if it has > 1 child.



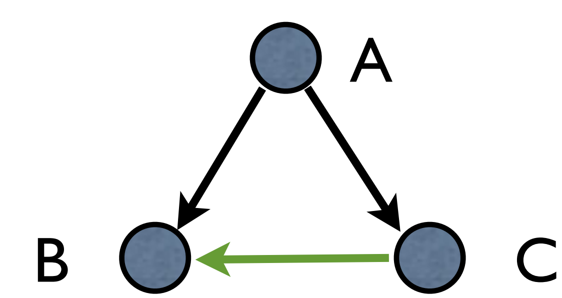


DFS Trees of Directed Graphs

- When we make a DFS of a directed graph, we still reach every node that is reachable from the start node.



- But it's no longer guaranteed that any or all of those nodes have paths back to the start point -- we no longer necessarily have a connected component to search.



Strongly Connected Components

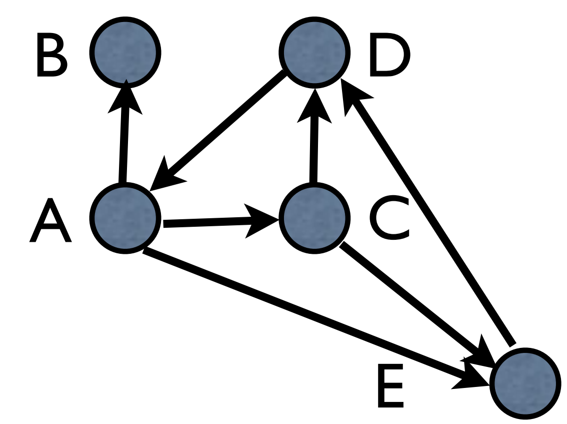
- Problem 9.6.2 (not on HW this term) has you work out how to use the DFS algorithm to find the **strongly connected components** of a directed graph -- the equivalence classes of the equivalence relation $P(x, y) \wedge P(y, x)$.
- If there is a back edge from a node t to an ancestor u, then all the nodes on the tree path from u down to t are in the same strongly connected component because they lie on a directed cycle.

DFS of a Directed Graph

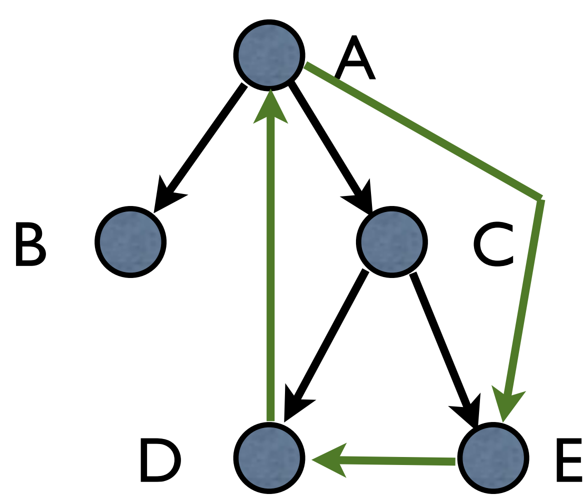
- In a directed graph we can no longer guarantee that all the edges are either tree edges or back edges -- what are the other possibilities?
- Let (u, v) be an arbitrary edge in a directed graph G. In what different ways could (u, v) be encountered in a DFS of G?

Tree and Forward Edges

- If we find u before v and first find v through the edge (u, v), it is a **tree edge**. e.g., (A, C)

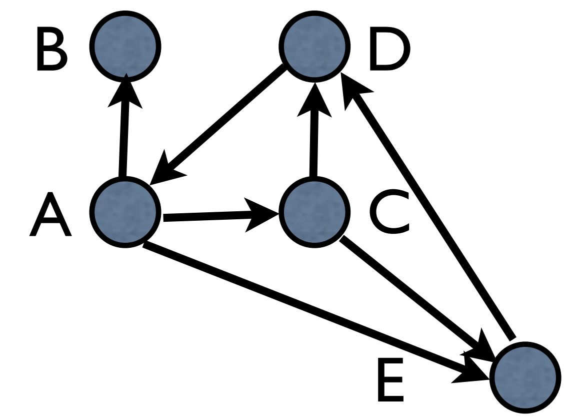


- If we find u before v, but find v through one of its siblings before we look at the edge (u, v), then (u, v) becomes a **forward edge** from u to a descendant. e.g., (A, E)

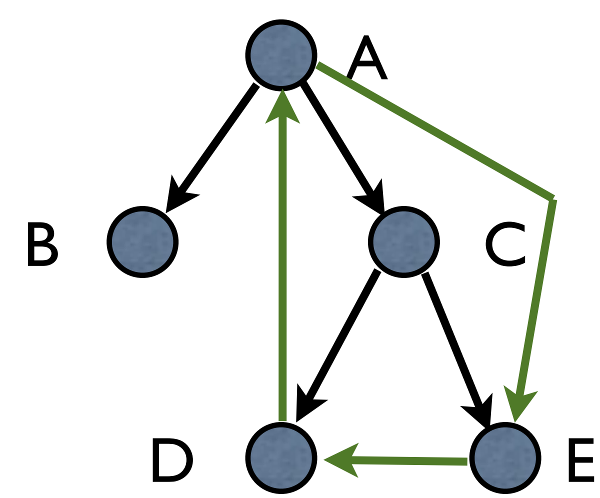


Back and Cross Edges

- If we find v before u, and find u while we are still processing v, then the edge (u, v) becomes a **back edge** just as in the undirected case. e.g., (D, A)

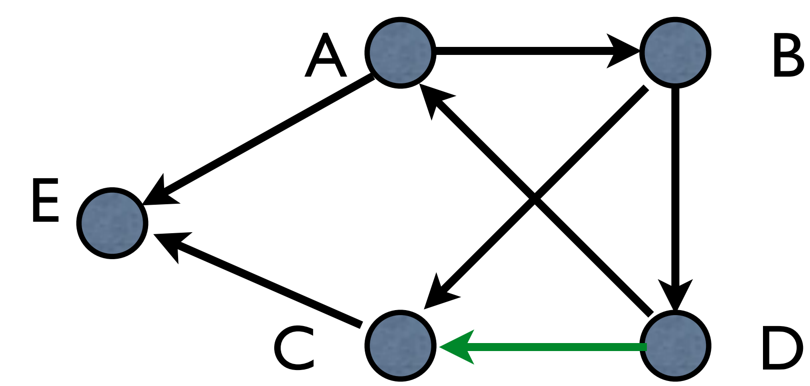


- If we find v before u and finish v before finding u (because there is no path from v to u), then (u, v) becomes a **cross edge**. e. g., (E, D)



Clicker Question #2

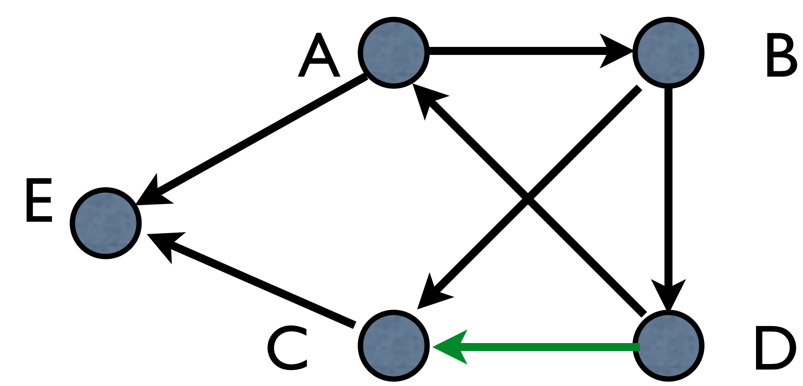
- What type of edge will the green edge become, if we do a DFS from A and always take neighbors alphabetically?
- (a) back edge
- (b) cross edge
- (c) forward edge
- (d) tree edge

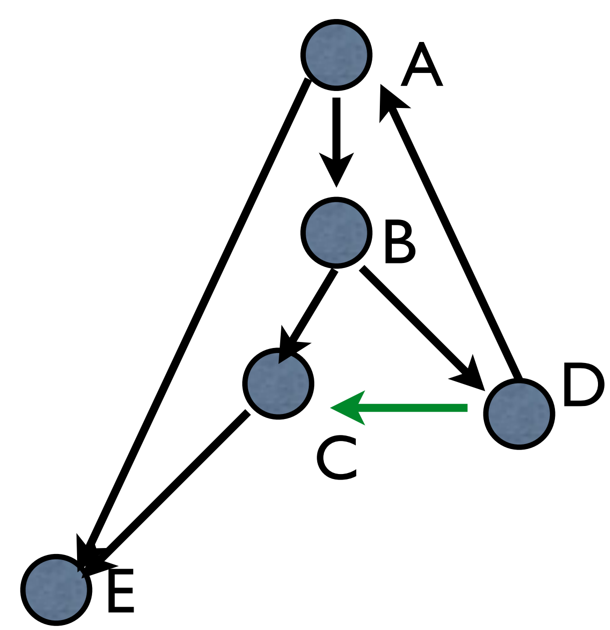


Not the Answer

Clicker Answer #2

- What type of edge will the green edge become, if we do a DFS from A and always take neighbors alphabetically?
- (a) back edge
- (b) cross edge
- (c) forward edge
- (d) tree edge





BFS Trees of Undirected Graphs

- A breadth-first search gives rise to tree edges in the same way -- (u, v) is a tree edge if we encounter v during the processing of u, and put v on the queue.
- The **BFS tree** is made up of all the tree edges, and is a rooted tree giving a shortest path (in number of edges) from the start node to each edge.
- If there are multiple shortest paths, the algorithm will choose one as the tree path.

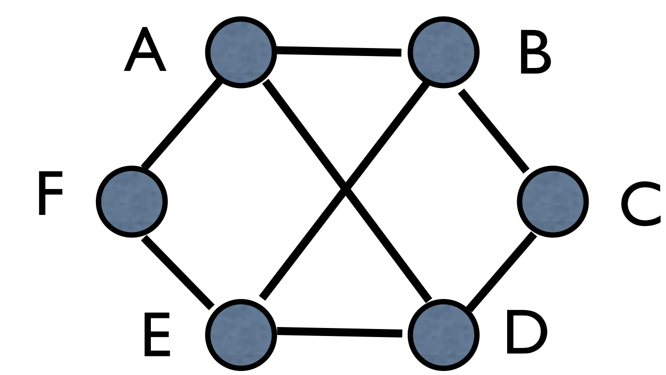
BFS Trees of Undirected Graphs

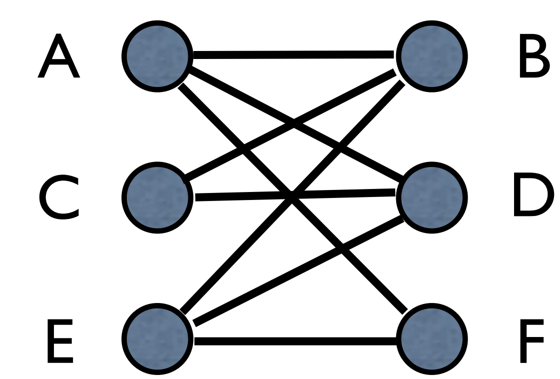
- If u is at level k of the tree, and (u, v) is a non-tree edge, we know that v has already been put on the queue before the edge is seen.
- If it is still on the queue, it must be at level k or k+1, because we are processing u at level k and there's a path from s to v of length k+1.
- If it has been finished, it must be at level k, because if it were < k (in an undirected graph) we would have already seen this edge going from v to u. (We explored all edges out of v when we took v off the queue.)

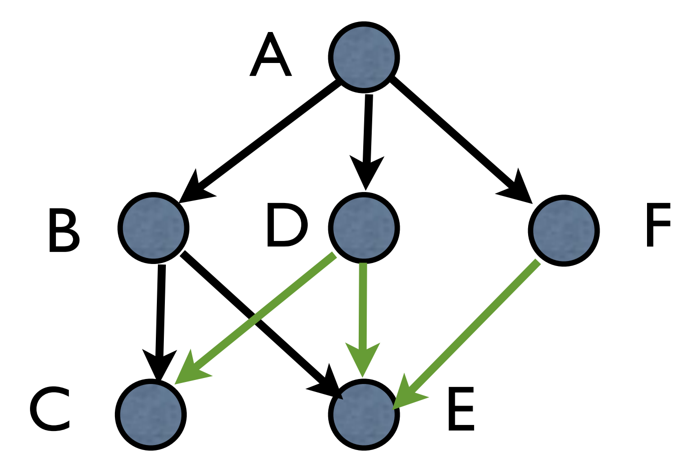
Bipartite Graphs

- An undirected graph is **bipartite** if and only if we never get an edge from one node to another at the same level.

- This follows from the theorem that an undirected graph is bipartite if and only if it has no **odd-length cycles**.)







Clicker Question #3

- Let G be a connected undirected graph. Which *one* of these conditions on G is equivalent to the statement that G is *not* a bipartite graph?
- (a) G has a cycle of even length.
- (b) In any DFS tree of G, every back edge goes up an odd number of levels.
- (c) In any BFS tree of G, there is a non-tree edge between two nodes at the same level.
- (d) There is a DFS tree of G with a back edge going up an odd number of levels.

Not the Answer

Clicker Answer #3

- Let G be a connected undirected graph. Which *one* of these conditions on G is equivalent to the statement that G is *not* a bipartite graph?
- (a) G has a cycle of even length.

there might also be an odd cycle

- (b) In any DFS tree of G, every back edge goes up an odd number of levels.

in this case G **is** bipartite

- (c) In any BFS tree of G, there is a non-tree edge between two nodes at the same level.
- (d) There is a DFS tree of G with a back edge going up an odd number of levels.

there might also be an edge going up an even number

BFS Trees of Directed Graphs

- In a BFS of a directed graph, the BFS tree will arrange the nodes into levels, based on their shortest-path distance from the start node (where again “shortest” means “fewest edges”).
- If u is at level k and we find v for the first time while processing u, then (u, v) will be a tree edge and v will be at level $k + 1$.

BFS Trees of Directed Graphs

- But if v has already been seen, it might be at *any* existing level of the tree from 0 to k or even k + 1, or might even not be in the tree at all!
- Remember that if a DFS or BFS finishes without reaching all the nodes, we start a new tree at a new start point. The node v might be in an earlier tree (which didn't contain a path to u), but still have an edge *from* u.