Graph Algorithms: Applications

CptS 223 – Advanced Data Structures

Larry Holder
School of Electrical Engineering and Computer Science
Washington State University

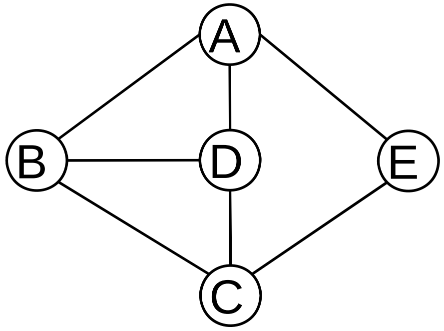
Applications

- Depth-first search
- Biconnectivity
- Euler circuits
- Strongly-connected components

Depth-First Search

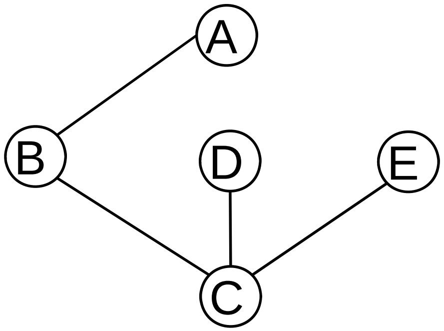
- Recursively visit every vertex in the graph
- Considers every edge in the graph
  - Assumes undirected edge (u,v) is in u's and v's adjacency list
- Visited flag prevents infinite loops
- Running time O(|V|+|E|)

```
DFS ()  ;; graph G=(V,E)
  foreach v in V
    if (! v.visited)
    then Visit (v)

Visit (vertex v)
  v.visited = true
  foreach w adjacent to v
    if (! w.visited)
    then Visit (w)
```



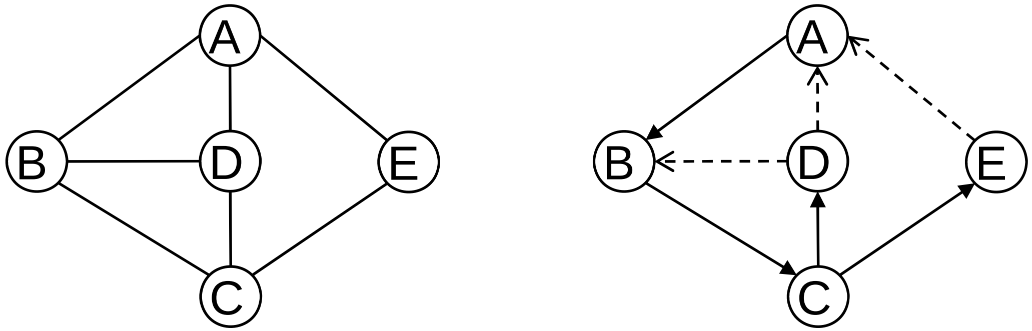
DFS Applications

- Undirected graph
  - Test if graph is connected
    - Run DFS from any vertex and then check if any vertices not visited
  - Depth-first spanning tree
    - Add edge (v,w) to spanning tree if w not yet visited (minimum spanning tree?)
    - If graph not connected, then depth-first spanning forest



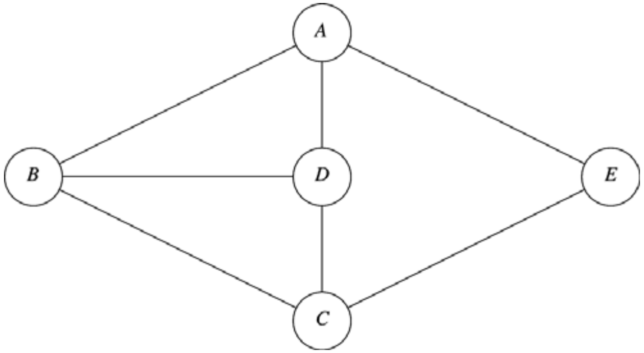
DFS Applications

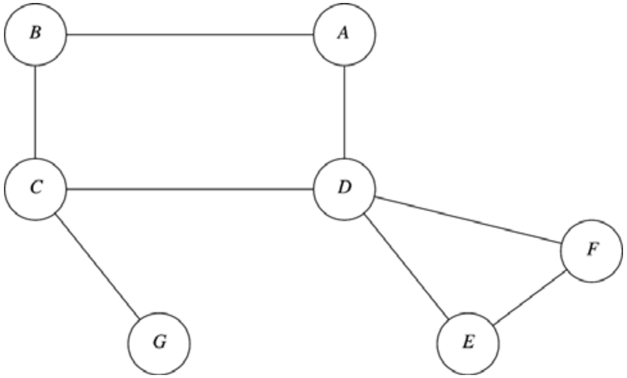
- Remembering the DFS traversal order is important for many applications
- Let the edges (v,w) added to the DF spanning tree be directed
- Add a directed back edge (dashed) if
  - w is already visited when considering edge (v,w), and
  - v is already visited when considering reverse edge (w,v)



Biconnectivity

- A connected, undirected graph is biconnected if the graph is still connected after removing any one vertex
  - I.e., when a "node" fails, there is always an alternative route
- If a graph is not biconnected, the disconnecting vertices are called articulation points
  - Critical points of interest in many applications



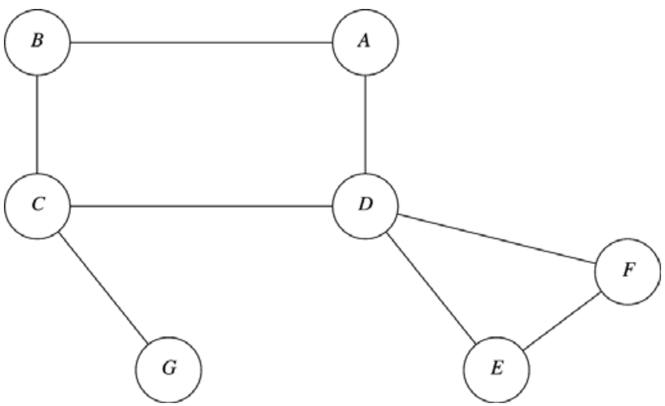


Biconnected?
Articulation points?

DFS Applications: Finding Articulation Points

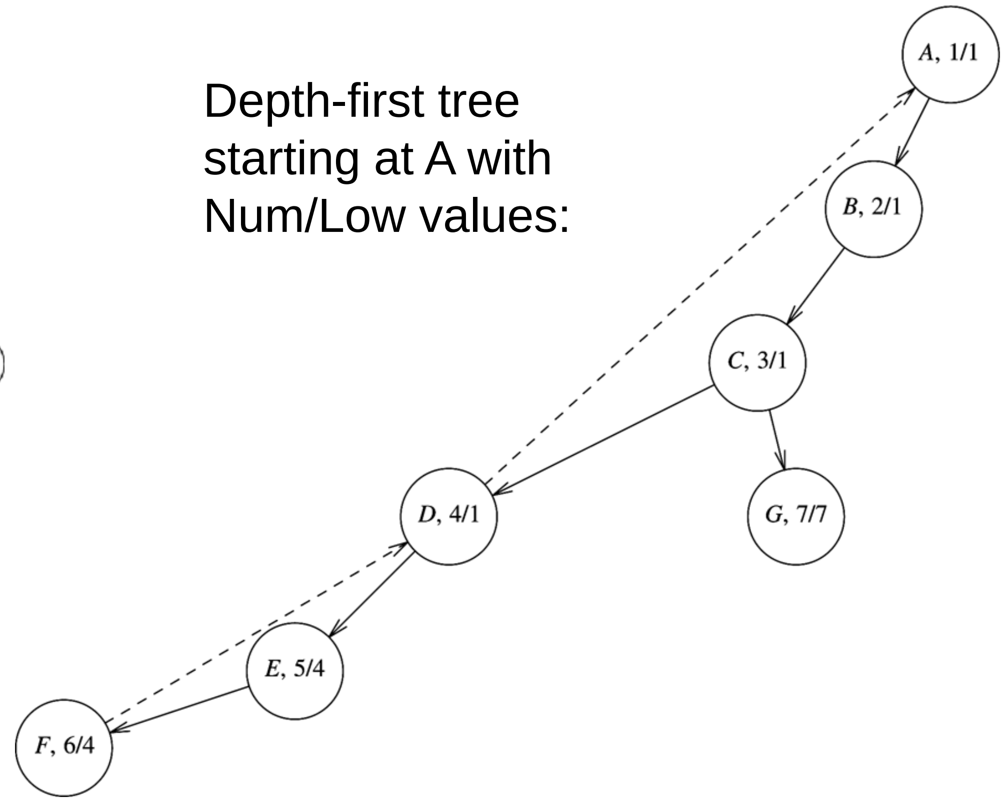
- From any vertex v, perform DFS and number vertices as they are visited
  - Num(v) is the visit number
- Let Low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then at most one back edge
  - Low(v) = minimum of
    - Num(v)
    - Lowest Num(w) among all back edges (v,w)
    - Lowest Low(w) among all tree edges (v,w)
- Can compute Num(v) and Low(v) in O(|E|+|V|) time

DFS Applications: Finding Articulation Points (Example)



Original Graph

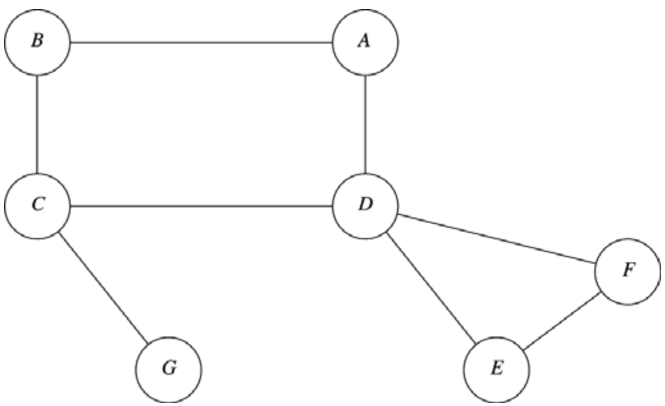
Depth-first tree starting at A with Num/Low values:



DFS Applications: Finding Articulation Points

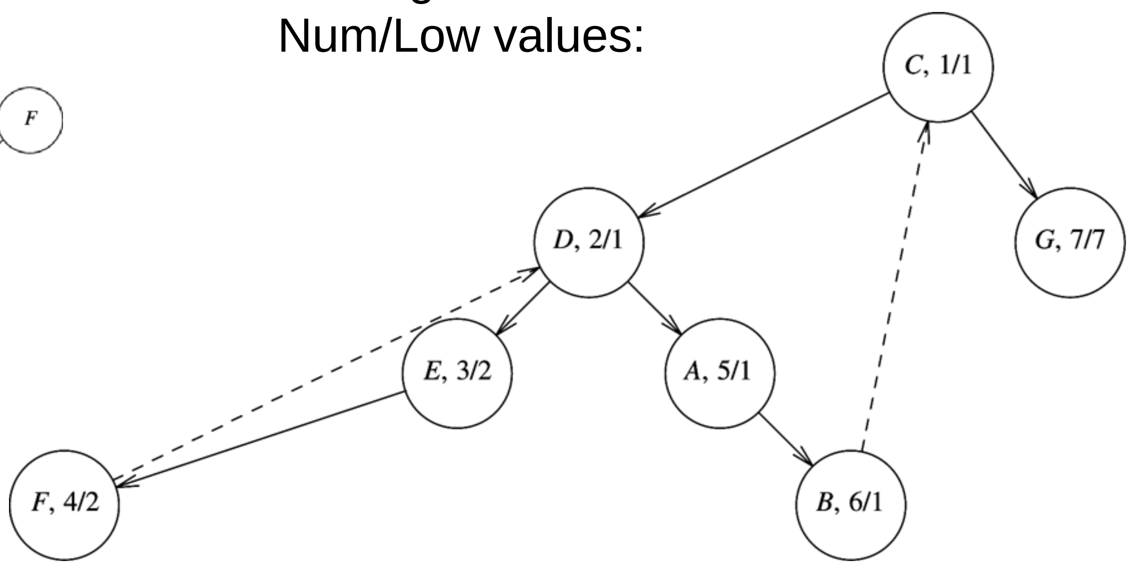
- Root is articulation point iff it has more than one child
- Any other vertex v is an articulation point iff v has some child w such that Low(w) $\geq$ Num(v)
  - I.e., is there a child w of v that cannot reach a vertex visited before v?
  - If yes, then removing v will disconnect w (and v is an articulation point)

DFS Applications: Finding Articulation Points (Example)



Original Graph

Depth-first tree starting at C with Num/Low values:



DFS Applications: Finding Articulation Points

- High-level algorithm
  - Perform pre-order traversal to compute Num
  - Perform post-order traversal to compute Low
  - Perform another post-order traversal to detect articulation points
- Last two post-order traversals can be combined
- In fact, all three traversals can be combined in one recursive algorithm

Implementation

```
/**
 * Assign num and compute parents.
 */
void Graph::assignNum( Vertex v )
{
    v.num = counter++;
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
        {
            w.parent = v;
            assignNum( w );
        }
}
```

```
/**
 * Assign low; also check for articulation points.
 */
void Graph::assignLow( Vertex v )
{
    v.low = v.num;  // Rule 1
    for each Vertex w adjacent to v
    {
        if( w.num > v.num )  // Forward edge
        {
            assignLow( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low );  // Rule 3
        }
        else
        if( v.parent != w )  // Back edge
            v.low = min( v.low, w.num );  // Rule 2
    }
}
```

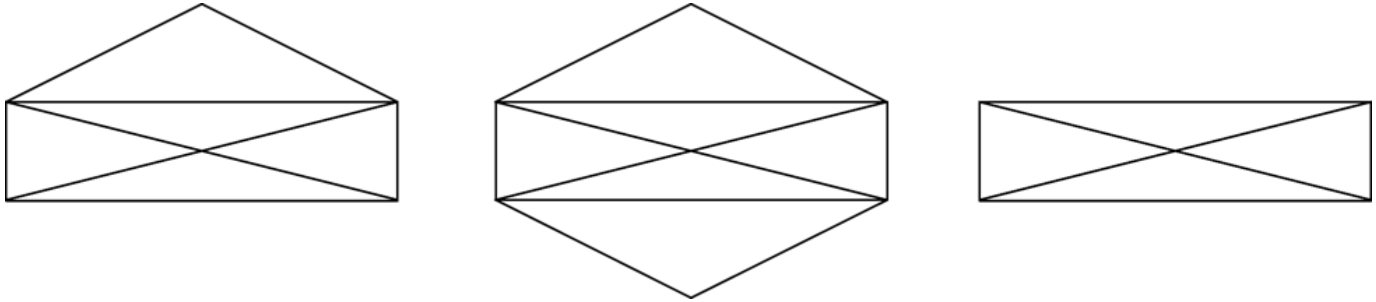
Check for root omitted.

```
void Graph::findArt( Vertex v )
{
    v.visited = true;
    v.low = v.num = counter++;  // Rule 1
    for each Vertex w adjacent to v
    {
        if( !w.visited )  // Forward edge
        {
            w.parent = v;
            findArt( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low );  // Rule 3
        }
        else
        if( v.parent != w )  // Back edge
            v.low = min( v.low, w.num );  // Rule 2
    }
}
```

Check for root omitted.

Euler Circuits

- Puzzle challenge
  - Can you draw a figure using a pen, drawing each line exactly once, without lifting the pen from the paper?
  - And, can you finish where you started?

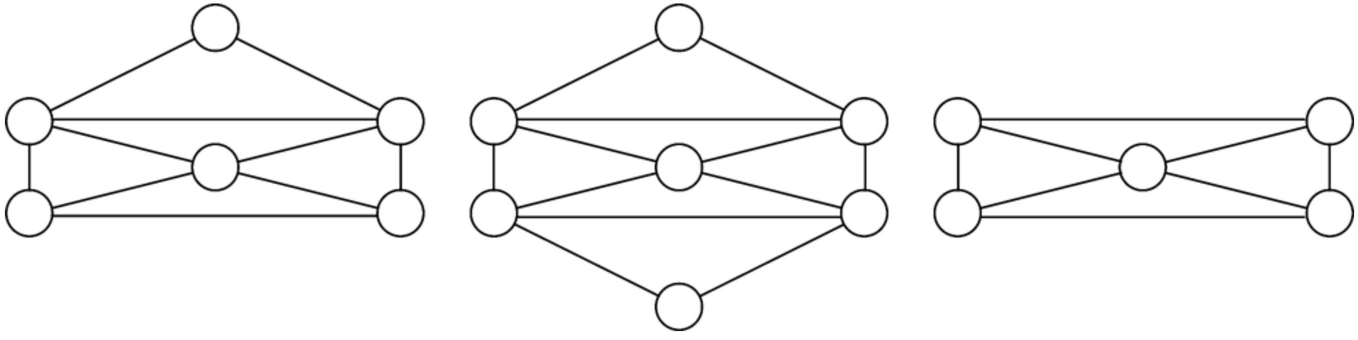


Euler Circuits

- Solved by Leonhard Euler in 1736 using a graph approach (DFS)
  - Also called an “Euler path” or “Euler tour”
  - Marked the beginning of graph theory

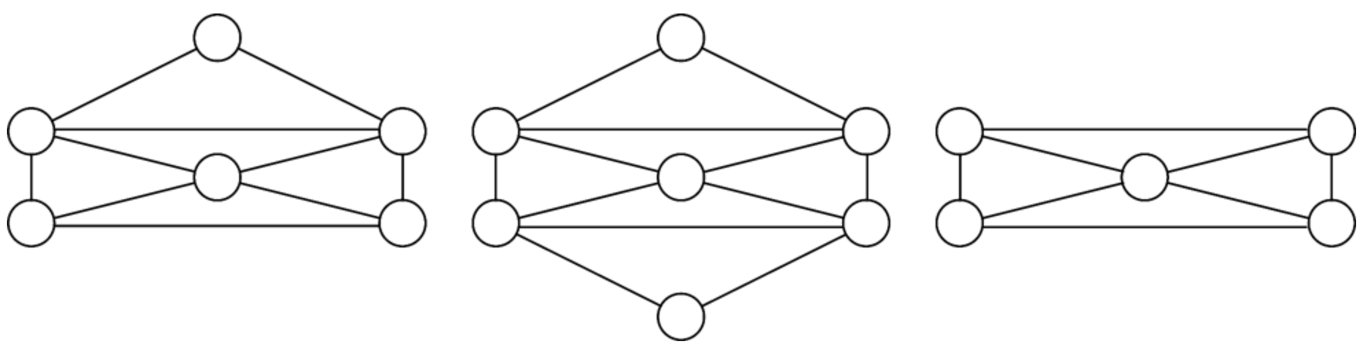
Euler Circuit Problem

- Assign a vertex to each intersection in the drawing
- Add an undirected edge for each line segment in the drawing
- Find a path in the graph that traverses each edge exactly once, and stops where it started



Euler Circuit Problem

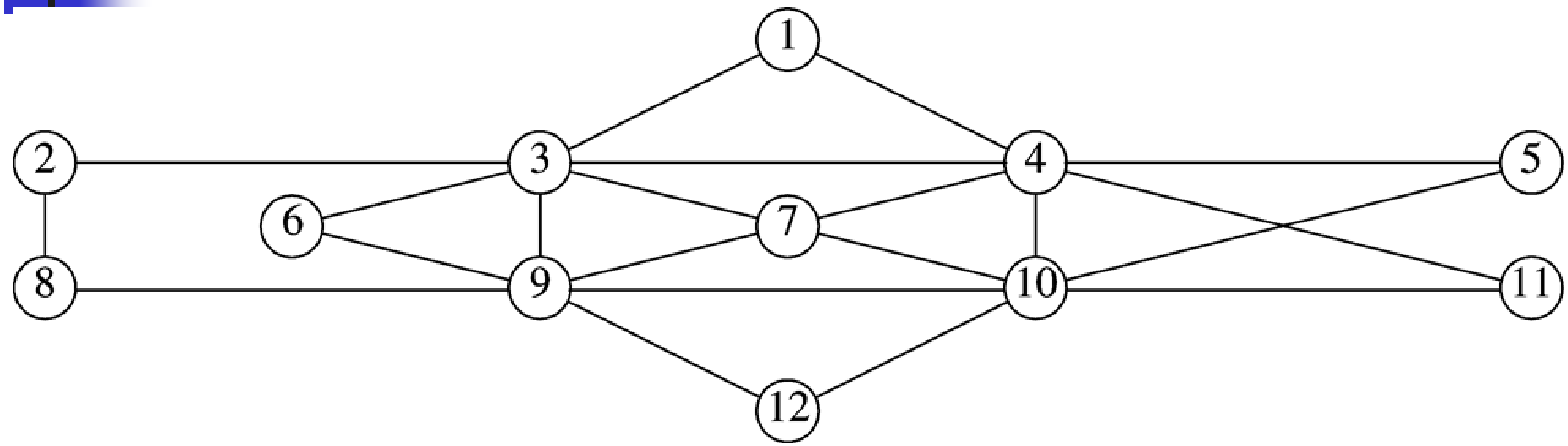
- Necessary and sufficient conditions
  - Graph must be connected
  - Each vertex must have an even degree
- Graph with two odd-degree vertices can have an Euler tour (not circuit)
- Any other graph has no Euler tour or circuit



Euler Circuit Problem

- Algorithm
  - Perform DFS from some vertex v until you return to v along path p
  - If some part of graph not included, perform DFS from first vertex v' on p that has an un-traversed edge (path p')
  - Splice p' into p
  - Continue until all edges traversed

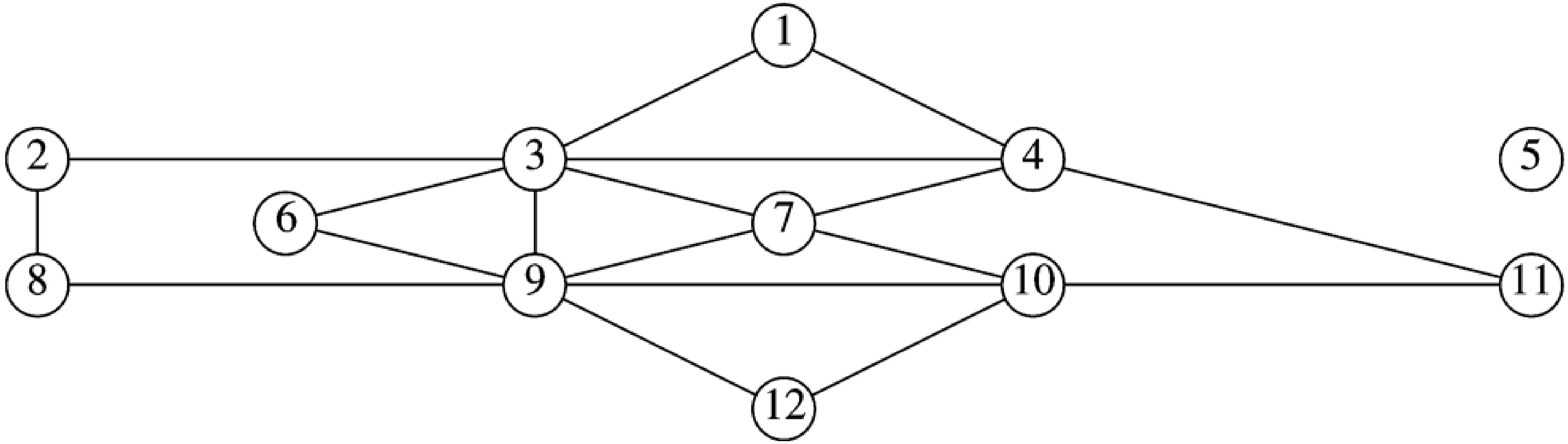
Euler Circuit Example



Start at vertex 5.
Suppose DFS visits 5, 4, 10, 5.

Euler Circuit Example (cont.)

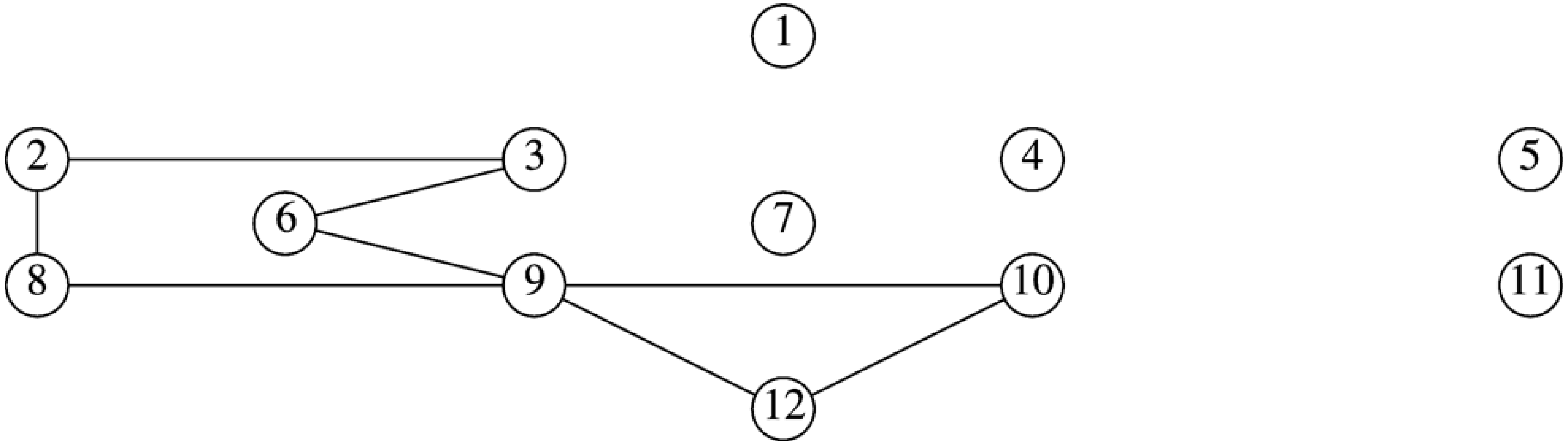
Graph remaining after 5, 4, 10, 5:



Start at vertex 4.
Suppose DFS visits 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4.
Splicing into previous path: 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

Euler Circuit Example (cont.)

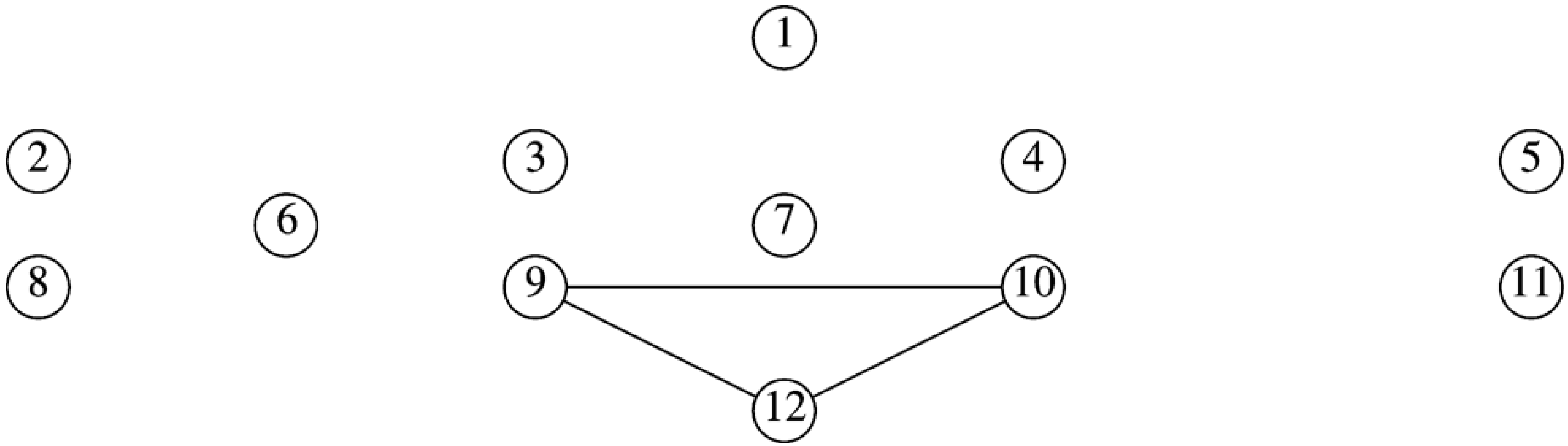
Graph remaining after 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5:



Start at vertex 3.
Suppose DFS visits 3, 2, 8, 9, 6, 3.
Splicing into previous path: 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

Euler Circuit Example (cont.)

Graph remaining after 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5:



Start at vertex 9.
Suppose DFS visits 9, 12, 10, 9.
Splicing into previous path: 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.
No more un-traversed edges, so above path is an Euler circuit.

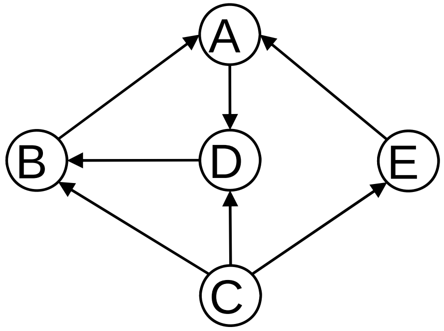
Euler Circuit Algorithm

- Implementation details
  - Maintain circuit as a linked list to support O(1) splicing
  - Maintain index on adjacency lists to avoid repeated searches for un-traversed edges
- Analysis
  - Each edge considered only once
  - Running time is $O(|E|+|V|)$

DFS on Directed Graphs

- Same algorithm
- Graph may be connected, but not strongly connected
- Still want the DF spanning forest to retain information about the search

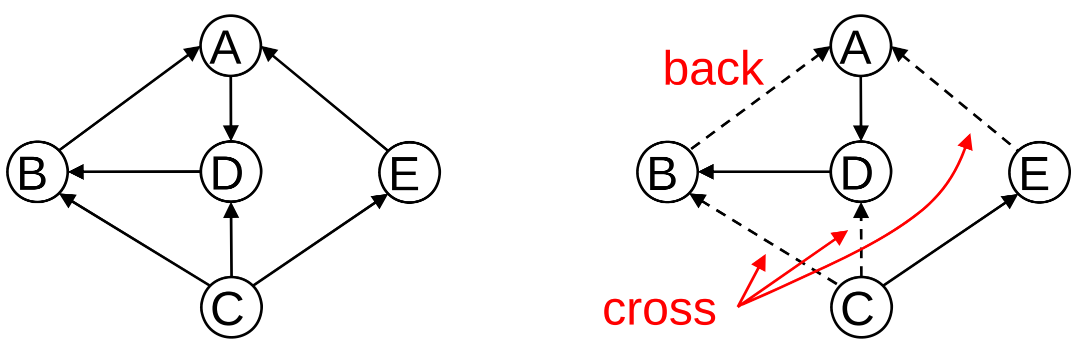
```
DFS ()  ;; graph G=(V,E)
  foreach v in V
    if (! v.visited)
    then Visit (v)

Visit (vertex v)
  v.visited = true
  foreach w adjacent to v
    if (! w.visited)
    then Visit (w)
```



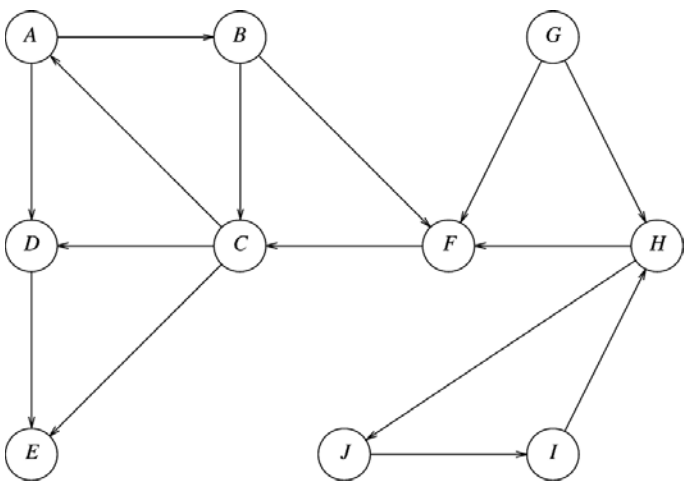
DF Spanning Forest

- Three types of edges in DF spanning forest
  - <u>Back edges</u> linking a vertex to an ancestor
  - <u>Forward edges</u> linking a vertex to a descendant
  - <u>Cross edges</u> linking two unrelated vertices

Graph: DF Spanning Forest:

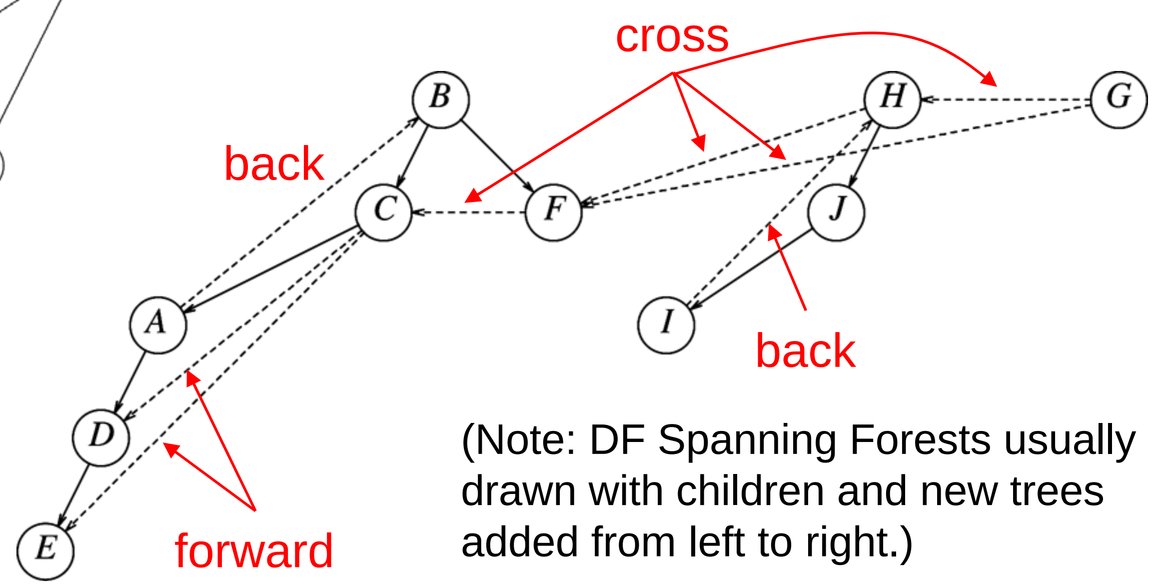


DF Spanning Forest



Graph

DF Spanning Forest



(Note: DF Spanning Forests usually drawn with children and new trees added from left to right.)

DFS on Directed Graphs

- Applications
  - Test if directed graph is acyclic
    - Has no back edges
  - Topological sort
    - Reverse post-order traversal of DF spanning forest

Strongly-Connected Components

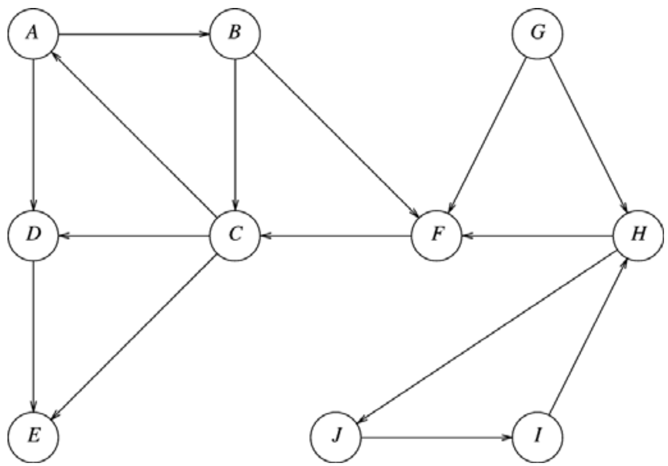
- A graph is strongly connected if every vertex can be reached from every other vertex
- A strongly-connected component of a graph is a subgraph that is strongly connected
- Would like to detect if a graph is strongly connected
- Would like to identify strongly-connected components of a graph
- Can be used to identify weaknesses in a network
- General approach: Perform two DFSs

Strongly-Connected Components

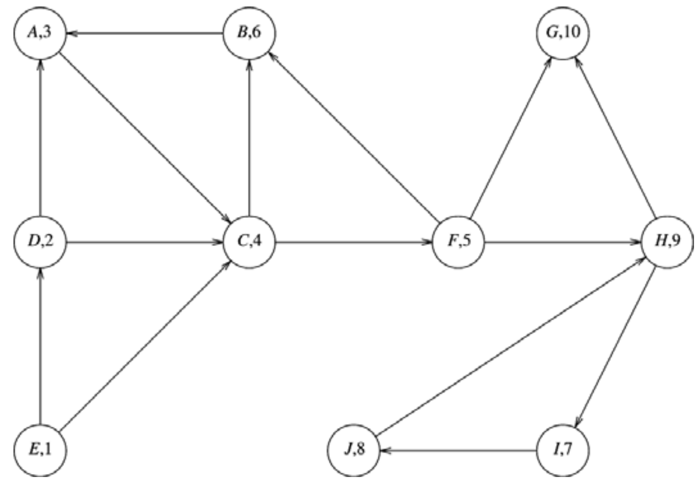
- Algorithm
  - Perform DFS on graph G
    - Number vertices according to a post-order traversal of the DF spanning forest
  - Construct graph G_r by reversing all edges in G
  - Perform DFS on G_r
    - Always start a new DFS (initial call to Visit) at the highest-numbered vertex
  - Each tree in resulting DF spanning forest is a strongly-connected component

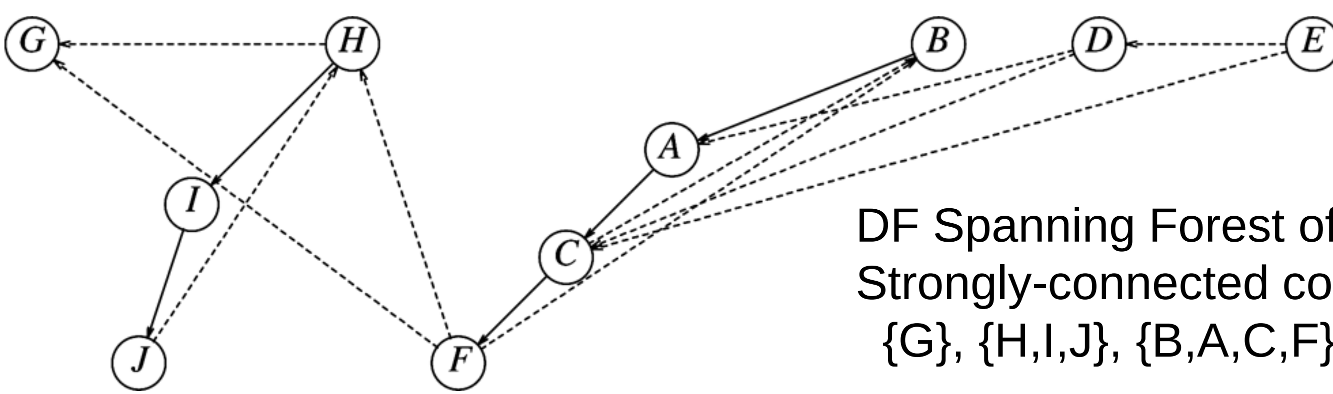
Strongly-Connected Components

Graph G



Graph G_r





DF Spanning Forest of G_r

Strongly-connected components:

{G}, {H,I,J}, {B,A,C,F}, {D}, {E}

Strongly-Connected Components: Analysis

- Correctness
  - If v and w are in a strongly-connected component
  - Then there is a path from v to w and a path from w to v
  - Therefore, there will also be a path between v and w in G and G_r
- Running time
  - Two executions of DFS
  - $O(|E|+|V|)$

Summary

- Graphs one of the most important data structures
- Studied for centuries
- Numerous applications
- Some of the hardest problems to solve are graph problems
    - E.g., Hamiltonian (simple) cycle, Clique