COMP261
Algorithms and Data Structures
2024 Tri 1

Jyoti Sahni
jyoti.sahni@ecs.vuw.ac.nz
Office Hours (COMP261): AM414, Thursday 10:00 – 12:00

Test 3

- When: 5-6 pm Thursday, May 9, 2024

- Syllabus: Everything covered from Week 7 – Tuesday (May 7), Week 9

- No lecture at 1:00 pm on Thursday, May 9

- Previous year question papers - 2023, 2022 (some sample questions)

- Refer to [Test3Preparation.pdf]() at the course wiki

Recap: Centrality

Centrality algorithms are used to understand the roles of particular nodes in a graph and their impact on that network.

- Degree centrality – Baseline metric
- Closeness centrality – How central a node is to the group
- Between centrality – finding control points
- Ranking – Overall influence

Different centrality algorithms can produce significantly different results based on what they were created to measure.

Recap: Page Rank

All the centrality measures (covered till now) measure the direct influence of a node. Page rank measure the transitive influence of nodes (influence of neighbours and neighbours of neighbours)

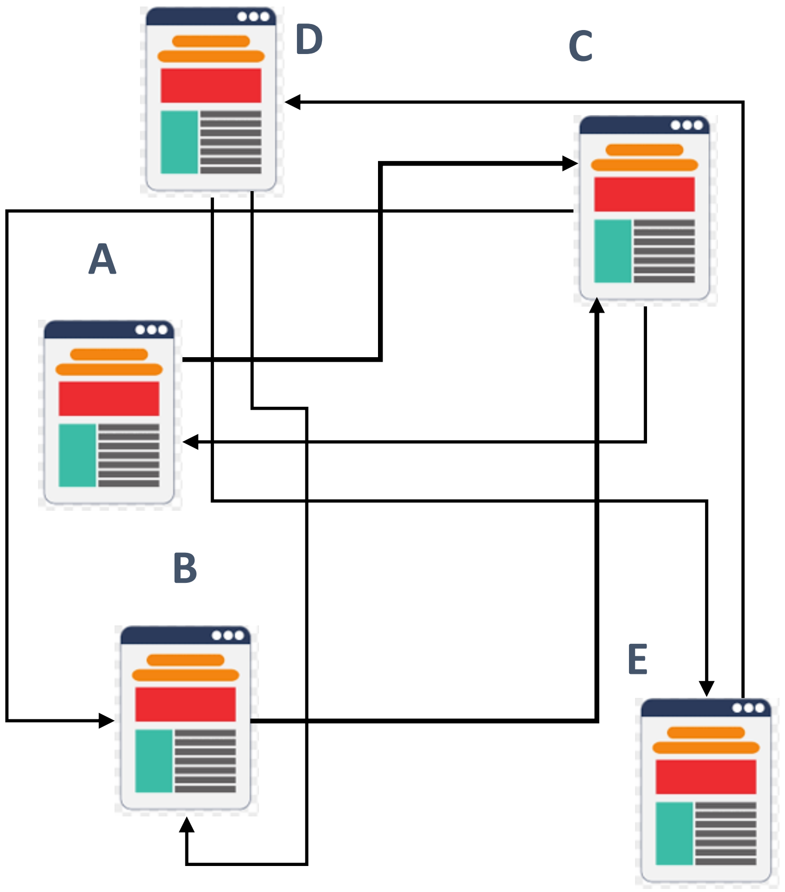
Page Rank, named after both “web page” and Google co-founder Larry Page, was the first algorithm that was used by Google to rank websites in their search engine results.

How PageRank works for the Google search engine: It counts the number and quality of links to a page to determine a rough estimate of how important the website is.

The underlying assumption is that a page with more incoming and more influential incoming links is more likely a credible source.

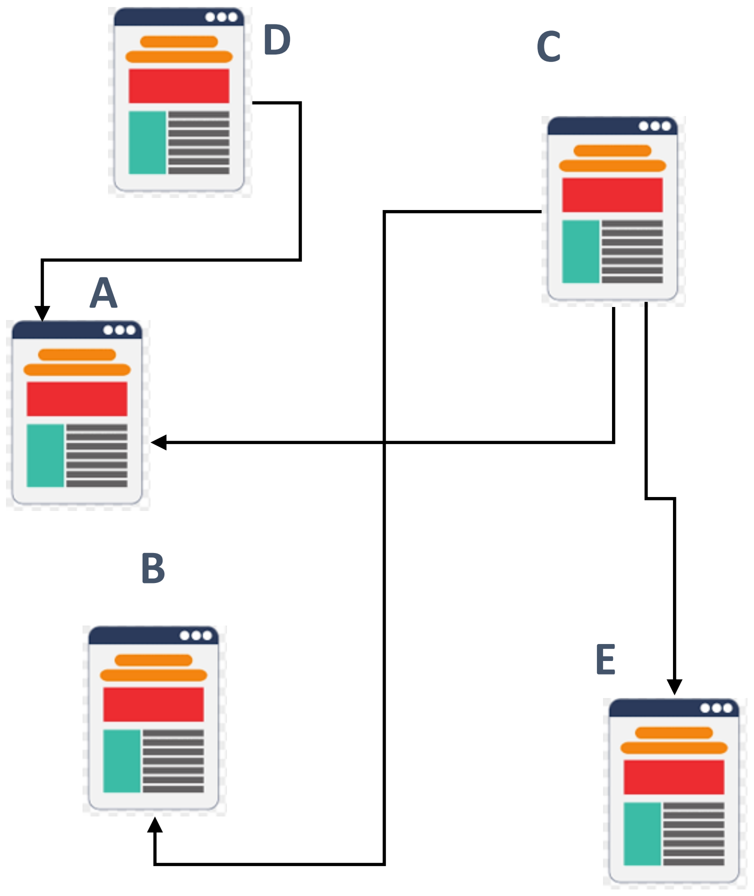
Recap: Some Preliminaries

- Backedge: If $page\ A$ links out to $page\ B$, then $page\ B$ is said to have a "backlink" from $page\ A$.
- $PR(P)$ — Each page P has a notion of its own page rank.
- $C(P)$ — Count of outgoing links for page P. Each page spreads its vote out evenly amongst all of it's outgoing links.
- We'll study a simplified version



Recap: Some Preliminaries

- Page rank of a page P is dependent on the page rank of the pages that have an outgoing link to P (nodes pointing to P)
- E.g. Page rank of A depends upon page ranks of C and D.
- The PageRank transferred from a given page to the targets of its outgoing links is divided equally among all outbound links.



Some Preliminaries: Page Rank

- **Random Surfer model:** Models how someone might browse the web without any particular goal in mind. The PageRank theory holds that an imaginary surfer who is randomly clicking on links will eventually stop clicking.
- **Damping factor:** The probability, at any step, that the person will continue following links is a damping factor d. The probability that they instead jump to any random page is $1 - d$.
- When calculating PageRank, pages with no outbound links are assumed to link out to all other pages in the collection.

$$PR(A) = \left(\frac{1-d}{N}\right) + d \sum_{B\ in\ inLinkNeighbours\ of\ A} \frac{PR(B)}{count(outbound\ links\ of\ B)} + \sum_{K\ with\ noOutLinks} \frac{PR(K)}{count(Number\ of\ Nodes\ in\ G)}$$

```
computePageRank(Graph graph, int iter, double dampingfactor){
   nNodes = get count of nodes in the graph
   //initialize
   for each node n in graph
      set $pageRank(n) = 1.0/nNodes$
   endFor
   $count = 1$
   Repeat
    $noOutLinkShare$ = 0
    for each node n in the graph that has no outlinks
       $noOutLinkShare = noOutLinkShare + dampingfactor \times (pageRank(n)/nNodes)$
    endFor
    for each node n in the graph
       $nRank = noOutLinkShare + (1 - dampingFactor)/nNodes$ // Page rank from
                                                              // random jumps
       neighboursShare = 0
       for each backneighbour b of n
          $neighboursShare = neighboursShare + pageRank(b)/\text{count of outedges of b}$
       EndFor
       $NewpageRank(n) = nRank + dampingFactor \times neighboursShare$
   endFor
   Update $pageRank$ with $NewpageRank$       //update page-rank for each page
                                             //with the newly computed values
   $count++;$
   Until $count > iter$
}
```

Cycles and Spanning Trees

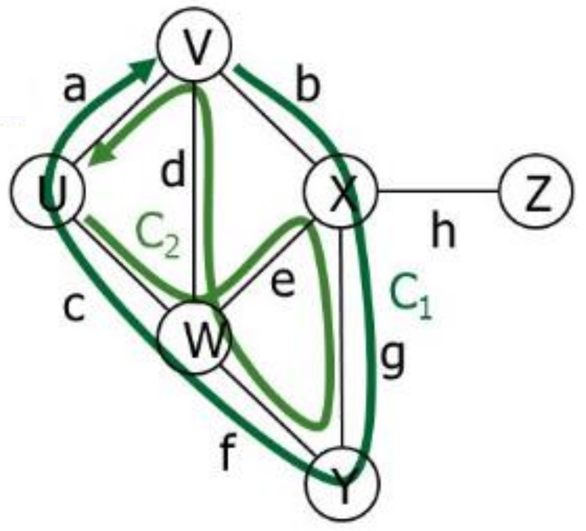
Cycle

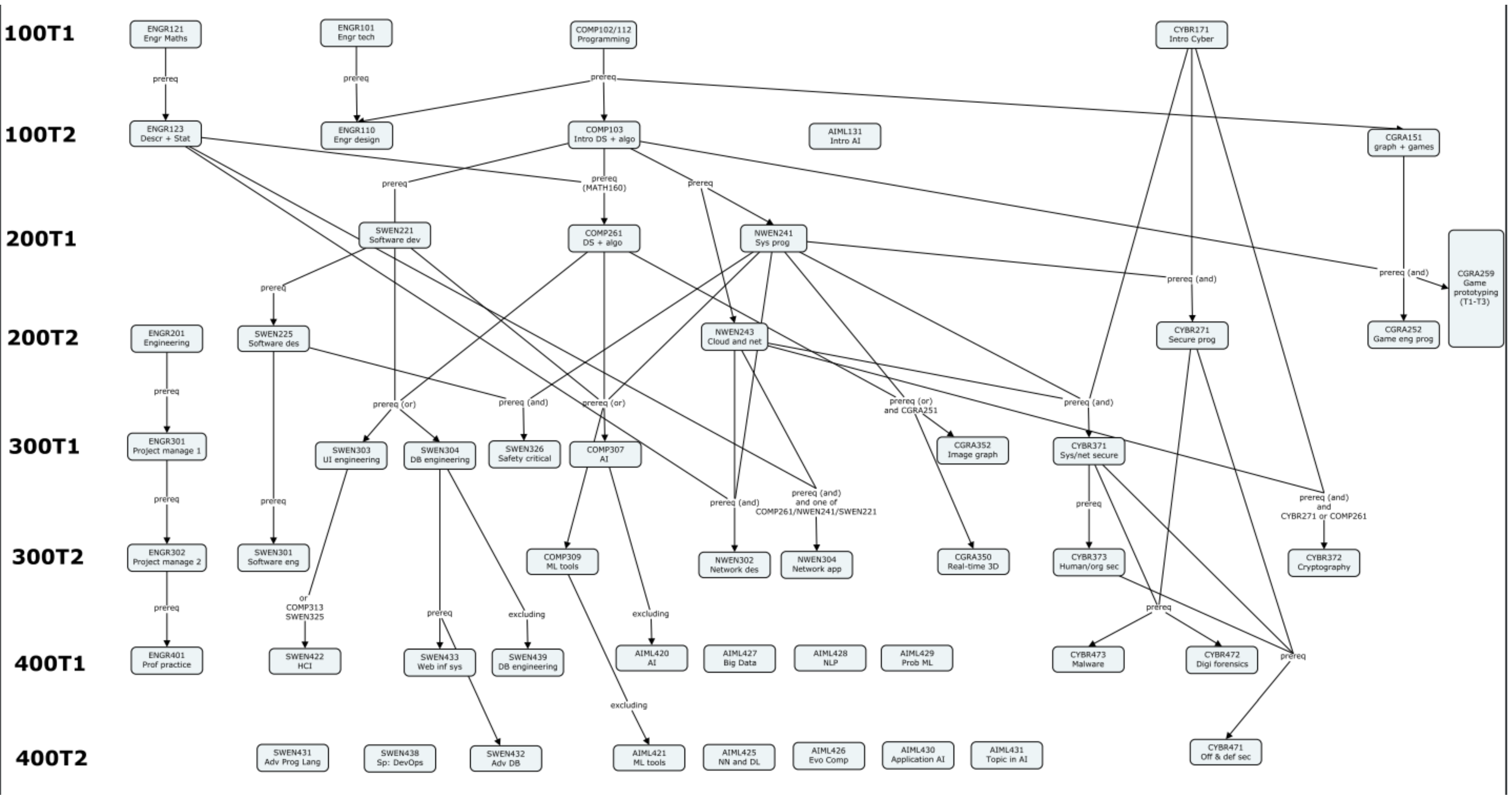
- A cycle in a graph is a non-empty path which begins and ends at the same vertex.

- A simple cycle is a cycle with no repeated vertices, except for the beginning and the ending vertex.

- $C_1 = (V, X, Y, W, U, V)$ is a simple cycle

- $C_2 = (U, W, X, Y, W, V, U)$ is a cycle that
- not simple





Courtesy: karsten lundqvist, ECS

Finding a Cycle in an Undirected Graph

- Many possible approaches: DFS based, BFS based and Topological sort based.

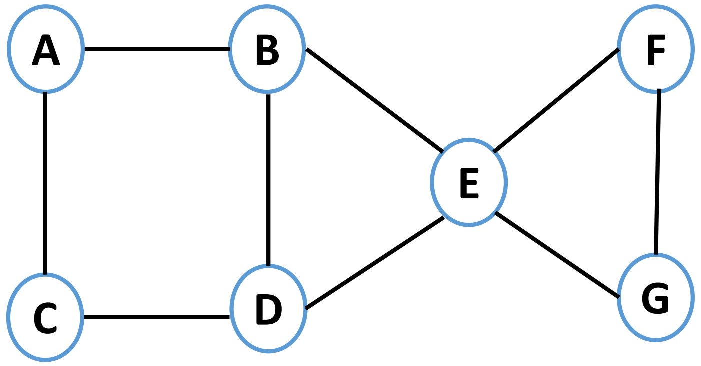
How DFS Works

- We start our search from a particular node.
- We then explore other nodes as long as we can go along a path.
- When reaching the end of the path, we do a backtrack up to the point where we began from and repeat the cycle until we have visited all nodes

DFS Pseudocode

```
$dfsTraversal(Graph\ G)$ {
  for each node n: n.visited = false
  for some node n: DFS(n)
}
DFS($Node\ n$) {
  n.marked = true
  print n
  for each j adjacent to n:
       if(!j.visited) {
              DFS(j)
    }
}
```

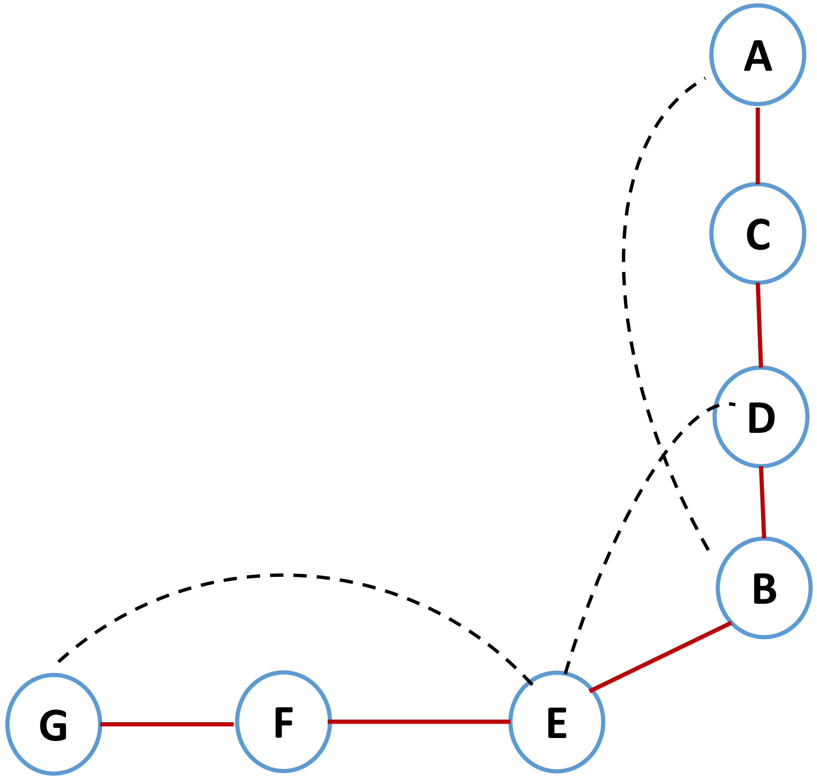
Example – DFS Traversal



0	A	$\rightarrow$ C, B
1	B	$\rightarrow$ A, D, E
2	C	$\rightarrow$ A, D
3	D	$\rightarrow$ C, B, E
4	E	$\rightarrow$ B, D, F, G
5	F	$\rightarrow$ E, G
6	G	$\rightarrow$ E, F

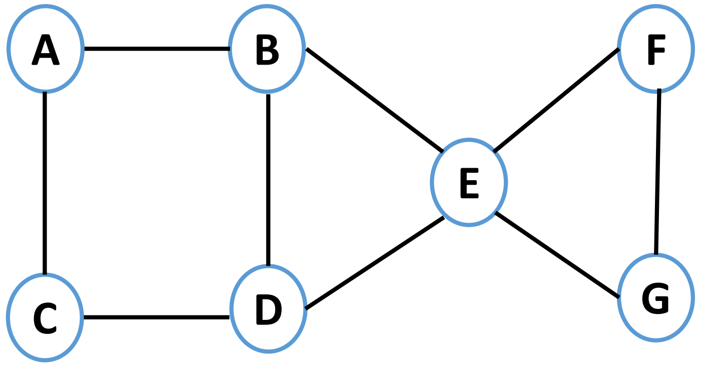
Visited

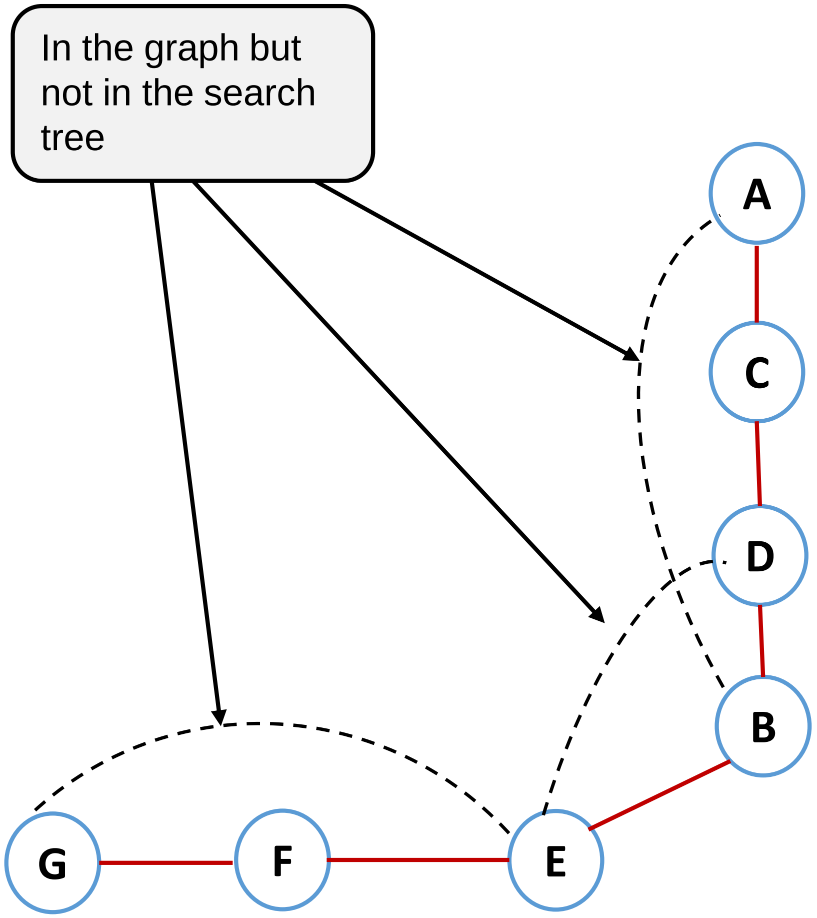
$\cancel{0}$ 1	$\cancel{0}$ 1	$\cancel{0}$ 1	$\cancel{0}$ 1	$\cancel{0}$ 1	$\cancel{0}$ 1	$\cancel{0}$ 1
A	B	C	D	E	F	G



You can also use a data member visited in the node definition. OR
Use a visited Set

Example – DFS Traversal

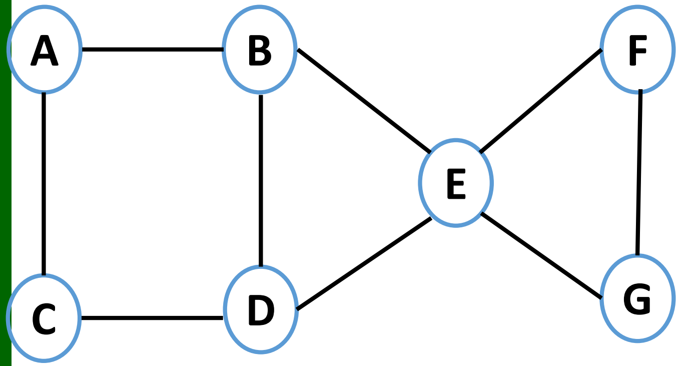




Back edge: Edge which is missing in the DFS tree but present in the graph

All the back edges which DFS skips over are part of cycles

Cycle Detection using DFS



- Steps:

- Start DFS traversal

- Keep track of parent of the node being visited

- If you find a node that has already been visited but is not the parent of the current node being visited – there is a cycle

0	A	$\rightarrow$ C, B
1	B	$\rightarrow$ A, D, E
2	C	$\rightarrow$ A, D
3	D	$\rightarrow$ C, B, E
4	E	$\rightarrow$ B, D, F, G
5	F	$\rightarrow$ E, G
6	G	$\rightarrow$ E, F

DFS (A, -1), visited(A) = true

DFS (C, A), visited(C) = true

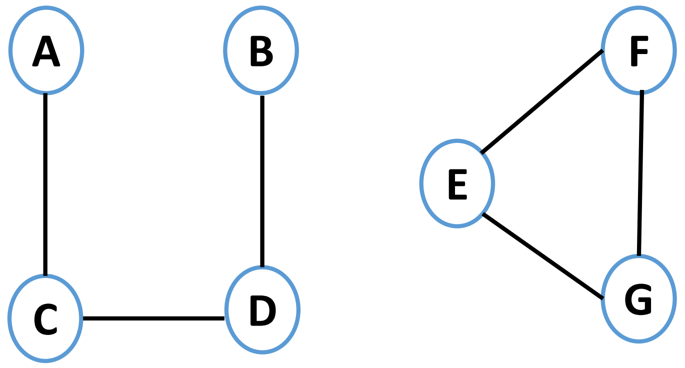
DFS (D, C), visited(D) =true

DFS (B, D), visited(B)=true

A has already been visited and A≠ parent(B). Cycle found

Cycle Detection using DFS

- What if the graph has multiple components?



- Repeat for each component until all nodes are visited or a cycle is found:
- Repeat the DFSCycle search from each unvisited node until all the nodes are visited or a cycle is found.

```
CycleDetection(Graph graph){
 nNodes = get count of nodes in the graph
 boolean visited[nNodes] // all initialized to false as none of the nodes
                        //have been visited
//Start DFS traversal
 for (v = 0; v < nNodes; v++) {
     if (!visited[v]) {
         if (isCyclicDFS(graph, v, visited, -1))
            return true;
         }
     }
     return false;
}
boolean isCyclicDFS(Graph graph, int v, boolean visited[], int parent) {
        visited[v] = true;
        for each neighbour n of node v {
            if (!visited[n]) {
                if (isCyclicDFS(graph, n, visited, v))
                    return true;
            } else if (n != parent) {
                // If the adjacent vertex is visited and is not parent of
                 //current vertex, then there is a cycle in the graph
                 return true;
            }
        }
        return false;
    }
```

Next Lecture

Spanning Trees