Design and Analysis of Algorithms

L15: Graphs
DFS & BFS

Dr. Ram P Rustagi
Sem IV (2020-Even)
Dept of CSE, KSIT
rprustagi@ksit.edu.in

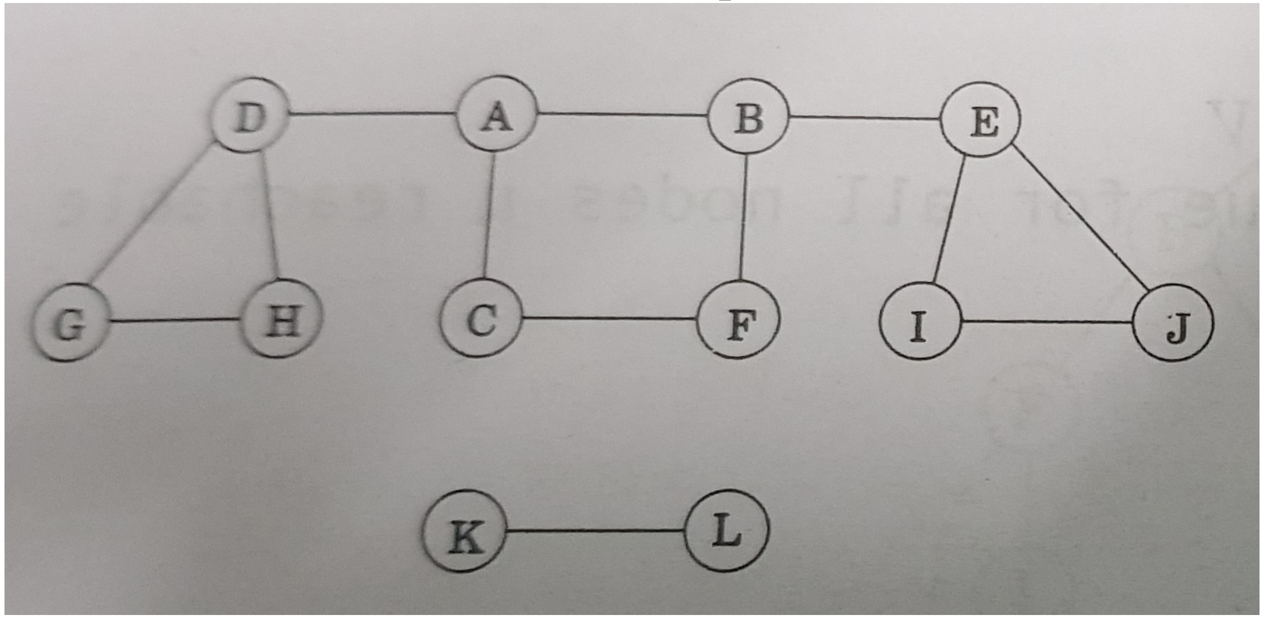
Resources

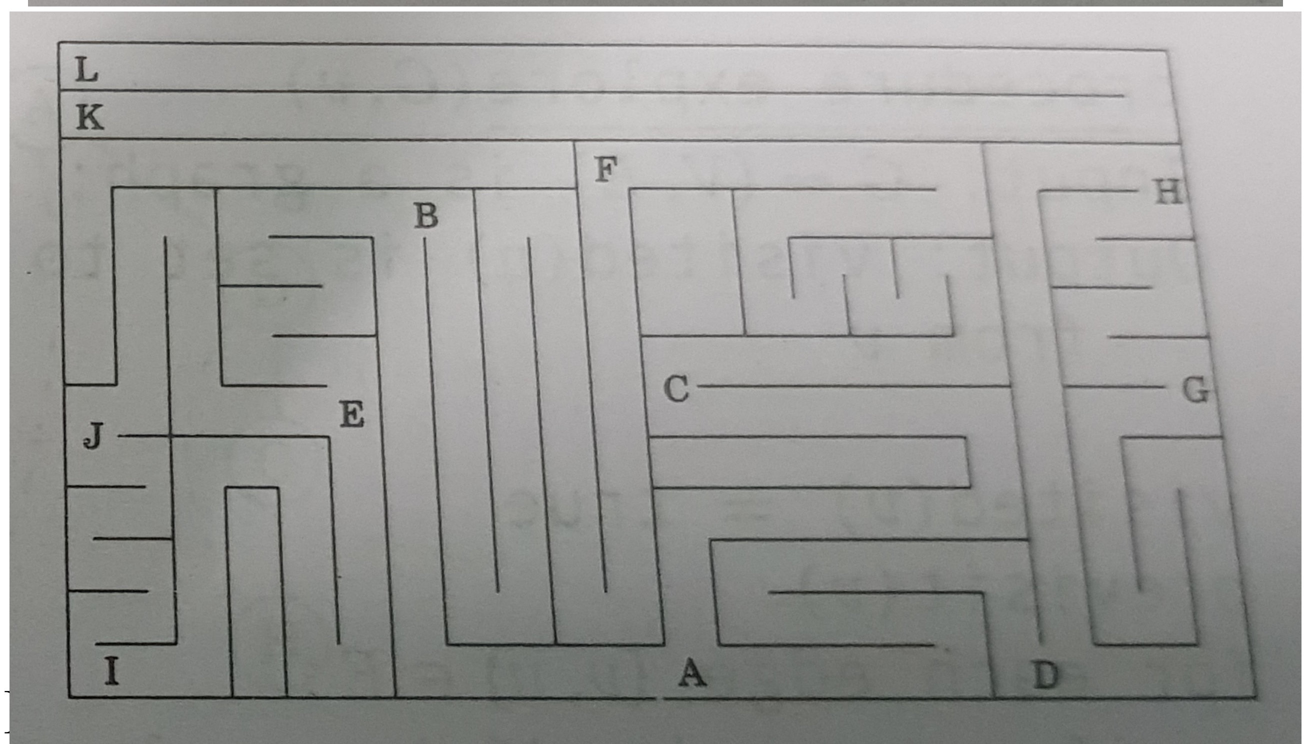
- Text book 1: Sec 5.1-5.3 - Levitin

Review of DFS and BFS

- Graph: set of nodes (vertices) connected by edges
  - Max number of edges are
    - undirected: `n(n-1)/2`; directed: `n(n-1)`
  - Assumption: no multiple edges b/w any two nodes.
  - Some pair of nodes may not have any edge
- Directed Graph
  - $A \rightarrow B$ is different than $B \rightarrow A$
- Implementation
  - Adjancey (Linked) list:
    - Each edge appears twice for undirected graph
  - Adjacency Matrix
    - Symmetric for undirected graph
    - Asymmetric for directed graph

Graph and Maze



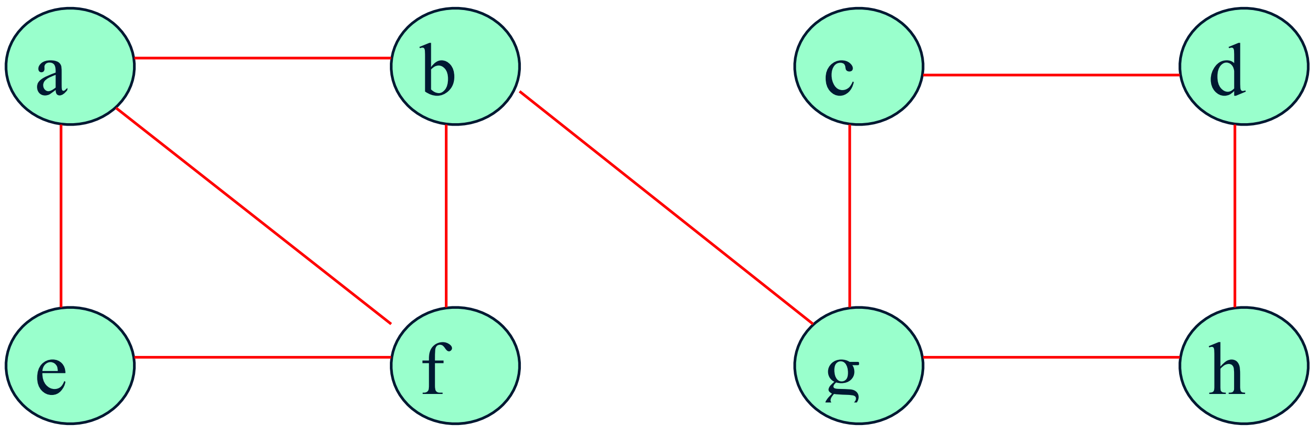


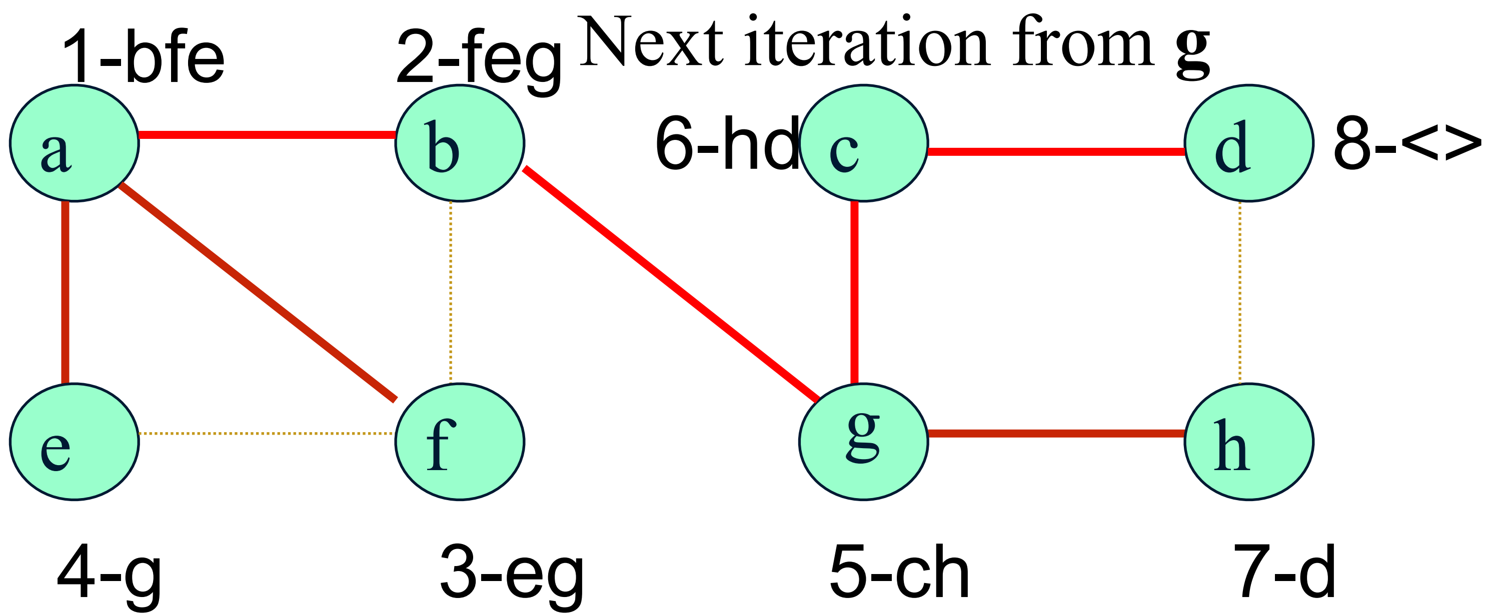
ref: Algorithms:
Dasgupta,
Vazirani,
Papadimitrou

BFS Algo (Undirected Graph)

```
proc BFS(v)
    mark(v) ← ++count
    Add v to queue.
    while queue is not empty do
        remove front vertex (i.e. v) from queue
        for each vertex w ∈ adjacency(v) do
            if w is marked with 0
                mark(w) ← ++count
                add w to the queue
#main
count←0; Initialize queue;
for each vertex v∈V do
    mark(v) ← 0
for each vertex v∈V do
    if mark(v) is 0
        BFS(v)
```

BFS Traversal





BFS Time Complexity

- Same efficiency as DFS
  - Adjacency matrices: $\Theta(|V|^2)$?
  - Adjacency lists: $\Theta(|V|+|E|)$?
- Vertices ordering
  - Single ordering of vertices
- Applications
  - Similar to DFS
  - Finding shortest path from a vertex to another becomes easier

BFS Traversal

- Visits graph vertices by
  - visiting all neighbours of last visited node
- Instead of a stack based implementation
  - Uses queue based implementation

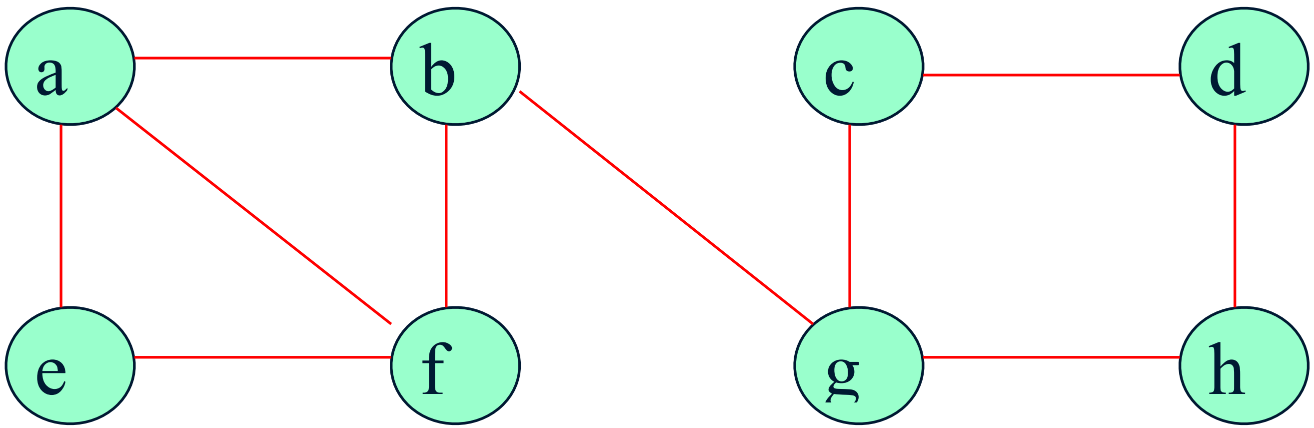
DFS

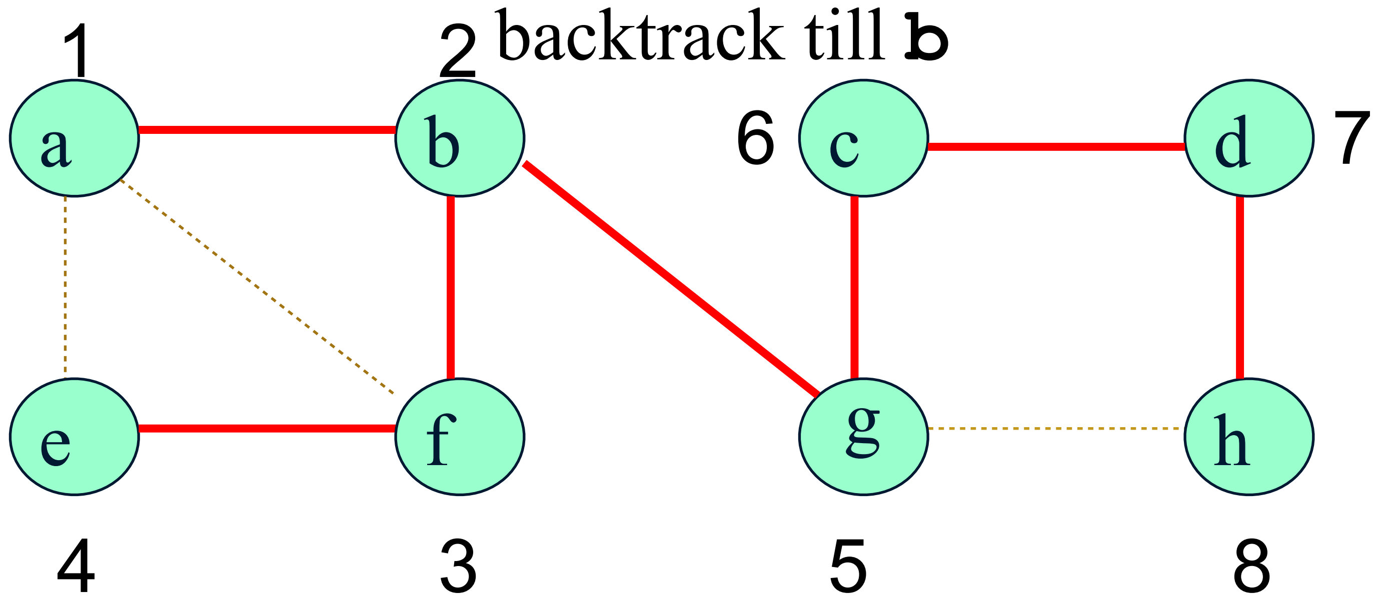
- DFS:
  - Start from a vertex (called root), mark it visited
  - Repeat the following
    - Find an unvisited vertex (not marked) connected by current node under consideration.
      - Mark this node as visited.
    - If there is no unvisited (unmarked) node connected to current node, backtrack.
- DFS Implementation
  - Using recursion (and stack)

DFS Algo

```
# Input: G=(V, E)
# o/p: nodes V marked in the order these are visited.
# mark of 0 implies unvisited.
proc dfs(v)
   mark(v) ← ++count;
   previsit(v)   //perform any Prework
   for each vertex w ∈ V adjacent to v do
      if w is marked with 0, then
         dfs(w);
   postvisit(v) // perform any Postwork
#end proc dfs(v)
for each vertex v ∈ V do
   mark(v) ← 0
count ← 0
for each vertex v ∈ V do
   if v is marked with 0, then
      dfs(v)
```

DFS Traversal





DFS Traversal: Time Complexity

- DFS implementation by Adjacency Matrix
  $\Theta(|V|^2)$
- DFS implementation by Adjacency Lists
  $\Theta(|V|+|E|)$
- Applications
  - Connected components
  - Checking for connected graph
  - Checking for acyclicity
  - Finding bi-connected components

Tree Traversal

- Forward Edge
- Cross Edge
- Back edge (Cycle)

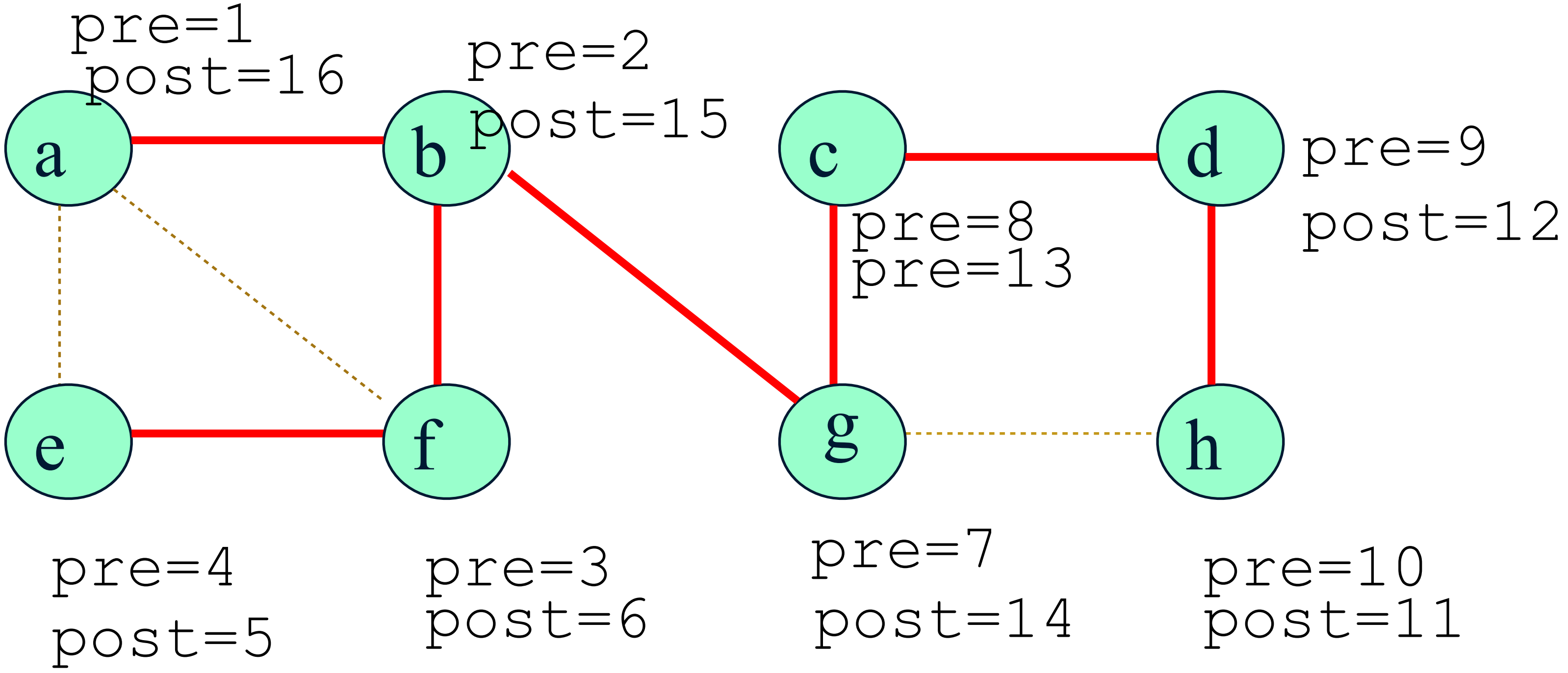
DFS: `previsit()` & `postvisit()`

- DFS uncovers the connectivity structure of graph
  - Takes linear time `O(|V|+|E|)`
- While exploring graph, some more can be done
  - note the time of first discovery (`previsit()`)
  - note the time of final departure (`postvisit()`)
  - Use counter as a clock which is initialized to 1

```
function previsit(v)
  pre[v] = clock++
function postvisit(v)
  post[v] = clock++
```

- Thus, for 8 node graph, each node gets two values

DFS Traversal



DFS: `previsit()` & `postvisit()`

- Property: for any two nodes u, v the
  - The intervals `[pre[u], post[u]]` and `[pre[v], post[v]]`
    - Are either disjoint, or
    - Contained in each other.
- Why this property
  - Node `[pre[u],post[u]]` represents the time the vertex `u` remains on stack
  - Example disjoint nodes: Node `f` and `g`
    `pre[f]=3`, `post[f]=6` & `pre[g]=7, post[g]=14`
  - Example nodes conained in each other: Node `g` and `c`
    `pre[g]=7`, `post[g]=14` & `pre[c]=8, post[g]=13`

Graph Terminology

- Graph nodes
  - *root*: the node where graph search starts
  - *descendant*: every other node is *descendant* of root
  - *ancestor*: if `v` is *descendant* of `u`, then `u` is *ancestor* of `v`
  - The family analogy accordingly has *child* and *parent*
- Undirected graph: *tree* edges, *non-tree* edges.
- Directed graph edges
    - *tree* edges: actual part of DFS traversal
    - *forward* edges: from node to non-child descendant
    - *back* edges: from node to an ancestor
    - *cross* edges: from node to neither an ancestor nor a descendant.
      - To a node which is completely explored

Directed Graph Terminology

- Consider two nodes `u` and `v`
  - `u` is ***ancestor*** of `v` if
    - `pre[u] < pre[v] < post[v] <post[u]`
  - `v` is ***descendant*** of `u` if
    - `pre[u] < pre[v] < post[v] <post[u]`
  - edge $u \rightarrow v$ is a tree/forward edge, if

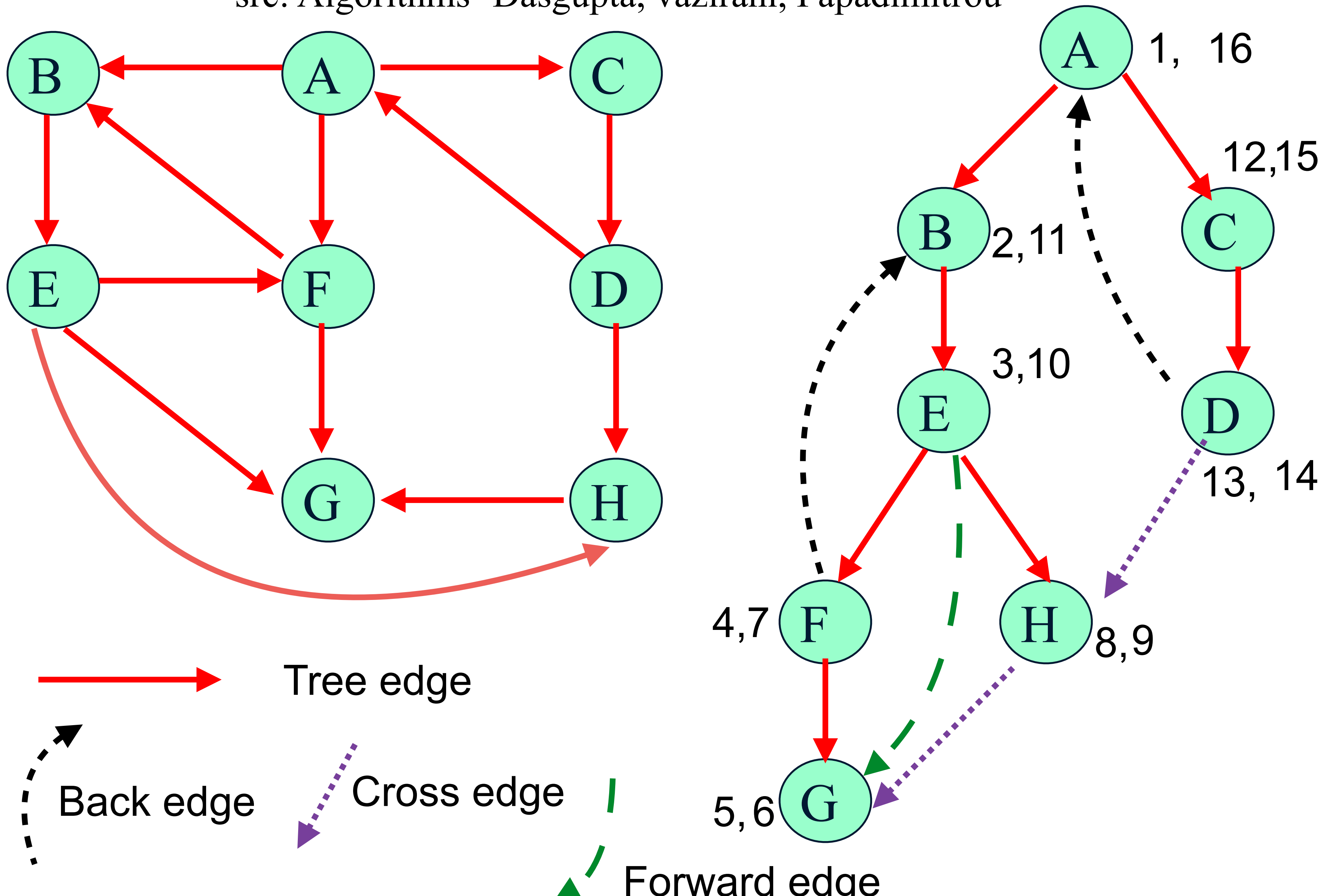
    $[_u \quad [_v \quad]_v \quad]_u$
  - edge $u \rightarrow v$ is a tree/forward edge, if

    $[_v \quad [_u \quad]_u \quad]_v$
  - edge $u \rightarrow v$ is a cross edge, if

    $[_v \quad]_v \quad [_u \quad]_u$

Directed Graph Edges

src: Algorithms- Dasgupta, vazirani, Papadimitrou



Summary

- Advantages and disadvantages of Divide and Conquer
- Decrease and conquer approach
- DFS traversal
- BFS traversal
  - Tree and forward edges
  - Cross edges
  - back edges