Introduction to Algorithms
Graph Algorithms

CSE 680
Prof. Roger Crawfis

Partially from *io.uwinnipeg.ca/~ychen2*

Graphs

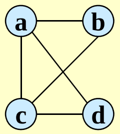
- *Graph G* = (*V*, *E*)
  - » *V* = set of vertices
  - » *E* = set of edges $\subseteq (V \times V)$
- Types of graphs
  - » Undirected: edge (*u*, *v*) = (*v*, *u*); for all *v*, (*v*, *v*) $\notin$ *E* (No self loops.)
  - » Directed: (*u*, *v*) is edge from *u* to *v*, denoted as $u \rightarrow v$. Self loops are allowed.
  - » Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbf{R}$.
  - » Dense: $|E| \approx |V|^2$.
  - » Sparse: $|E| << |V|^2$.
- $|E| = O(|V|^2)$

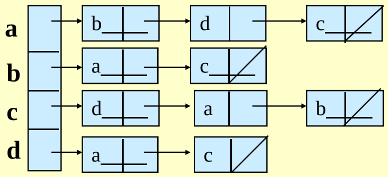
Graphs

- If $(u, v) \in E$, then vertex *v* is adjacent to vertex *u*.
- Adjacency relationship is:
  - » Symmetric if *G* is undirected.
  - » Not necessarily so if *G* is directed.
- If *G* is connected:
  - » There is a path between every pair of vertices.
  - » $|E| \geq |V| - 1$.
  - » Furthermore, if $|E| = |V| - 1$, then *G* is a tree.
- Other definitions in Appendix B (B.4 and B.5) as needed.

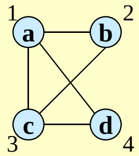
Representation of Graphs

- Two standard ways.
  - » Adjacency Lists.





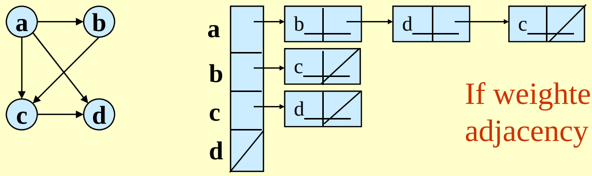
  - » Adjacency Matrix.



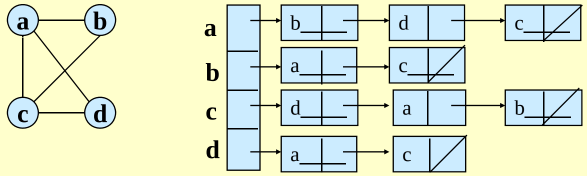
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Adjacency Lists

- Consists of an array *Adj* of |*V*| lists.
- One list per vertex.
- For $u \in V$, *Adj*[*u*] consists of all vertices adjacent to *u*.



If weighted, store weights also in adjacency lists.



Storage Requirement

- For directed graphs:
  - » Sum of lengths of all adj. lists is

$$\sum_{v \in V} \text{out-degree}(v) = |E|$$

No. of edges leaving *v*

  - » Total storage: $\Theta(|V| + |E|)$
- For undirected graphs:
  - » Sum of lengths of all adj. lists is

$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

No. of edges incident on *v*. Edge (*u*,*v*) is incident on vertices *u* and *v*.

  - » Total storage: $\Theta(|V| + |E|)$

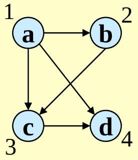
Pros and Cons: adj list

- Pros
  - » Space-efficient, when a graph is sparse.
  - » Can be modified to support many graph variants.
- Cons
  - » Determining if an edge $(u, v) \in G$ is not efficient.
    - Have to search in *u*'s adjacency list. Θ(degree(*u*)) time.
    - Θ(*V*) in the worst case.

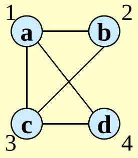
Adjacency Matrix

- $|V| \times |V|$ matrix *A*.
- Number vertices from 1 to |*V*| in some arbitrary manner.
- *A* is then given by:

$$A[i, j] = a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$



	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^T$ for undirected graphs.

Space and Time

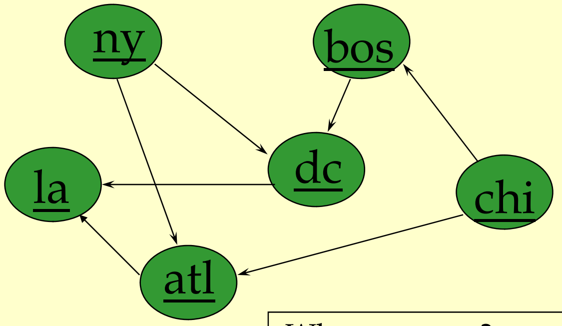
- **Space:** $\Theta(V^2)$.
  - » Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph.

Some graph operations

	adjacency matrix	adjacency lists
insertEdge	O(1)	O(e)
isEdge	O(1)	O(e)
#successors?	O(V)	O(e)
#predecessors?	O(V)	O(E)

---

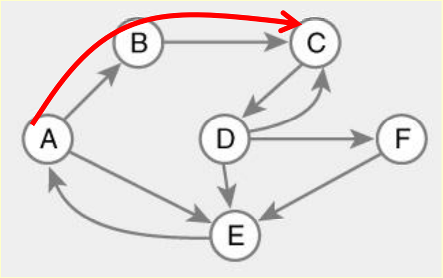
traversing a graph



Where to start?
Will all vertices be visited?
How to prevent multiple visits?

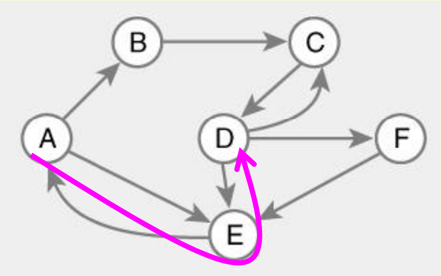
Graph Definitions

- Path
  - » Sequence of nodes $n_1, n_2, \dots n_k$
  - » Edge exists between each pair of nodes n_i, n_{i+1}
  - » Example
    - A, B, C is a path



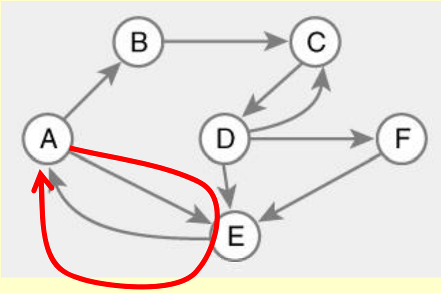
Graph Definitions

- Path
  - Sequence of nodes n_1, n_2, … n_k
  - Edge exists between each pair of nodes n_i , n_{i+1}
  - Example
    - A, B, C is a path
    - A, E, D is not a path



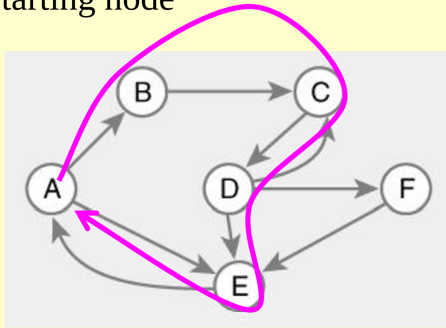
Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A



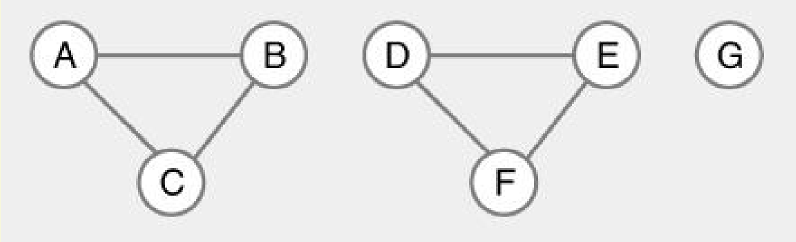
Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A
    - A, B, C, D, E, A
- Simple path
  - No cycles in path
- Acyclic graph
  - No cycles in graph



Graph Definitions

- Reachable
  - Path exists between nodes
- Connected graph
  - Every node is reachable from some node in graph



Unconnected graphs

Graph-searching Algorithms

- Searching a graph:
  - Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

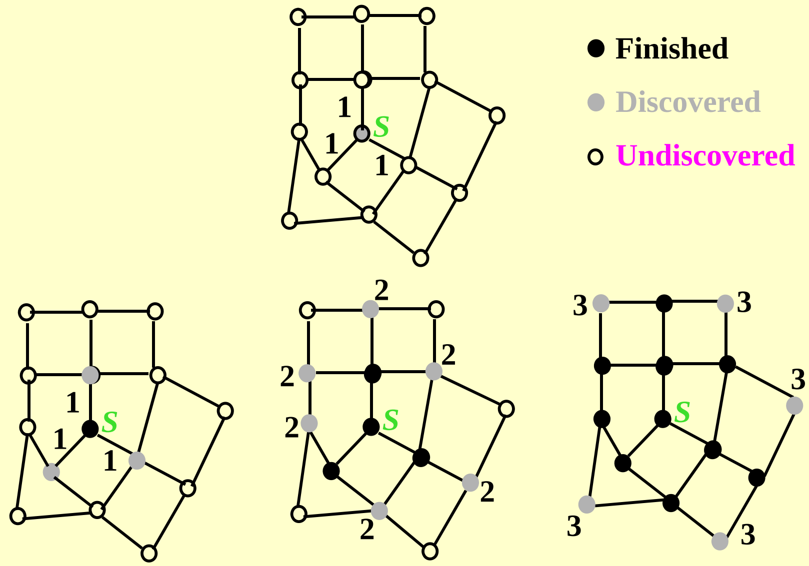
Breadth-first Search

- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.

BFS for Shortest Paths



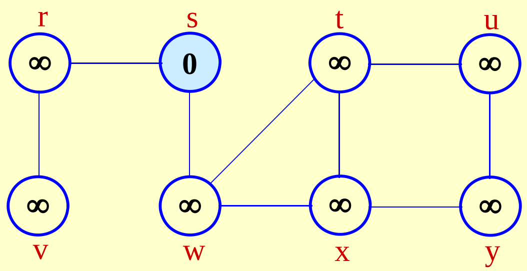
BFS(G,s)

```
for each vertex u in V[G] – {s}
    do color[u] ← white                 initialization
       d[u] ← ∞
       π[u] ← nil
color[s] ← gray
d[s] ← 0                                access source s
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
    do u ← dequeue(Q)
       for each v in Adj[u]
           do if color[v] = white
                 then color[v] ← gray
                      d[v] ← d[u] + 1
                      π[v] ← u
                      enqueue(Q,v)
       color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

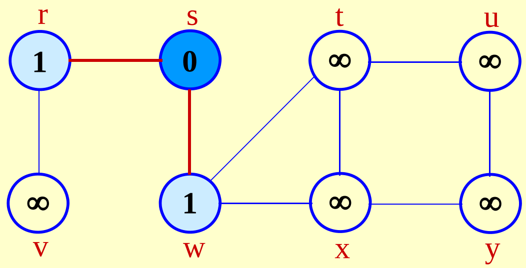
Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

Example (BFS)



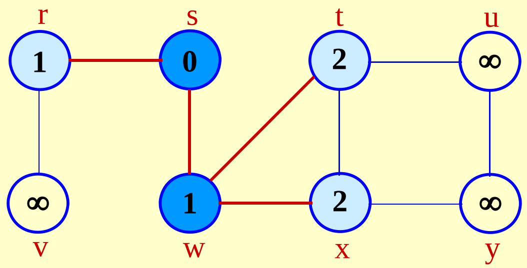
Q: s
0

Example (BFS)



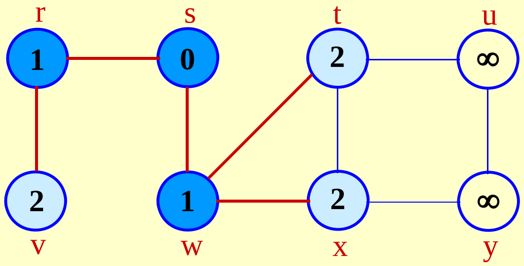
Q: w r
1 1

Example (BFS)



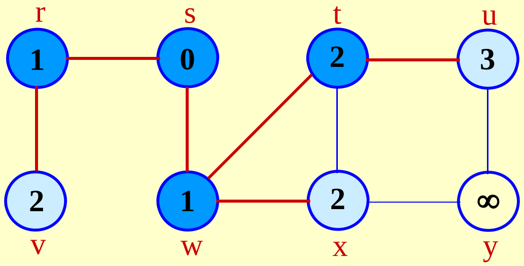
Q: r t x
1 2 2

Example (BFS)



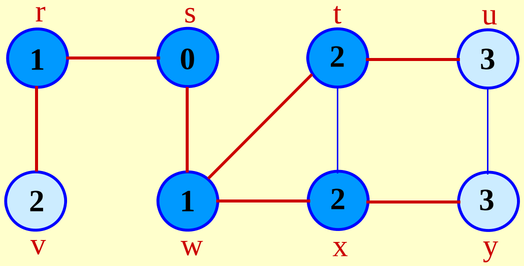
Q: t x v
2 2 2

Example (BFS)



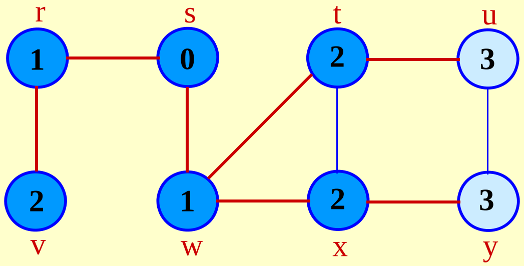
Q: x v u
2 2 3

Example (BFS)



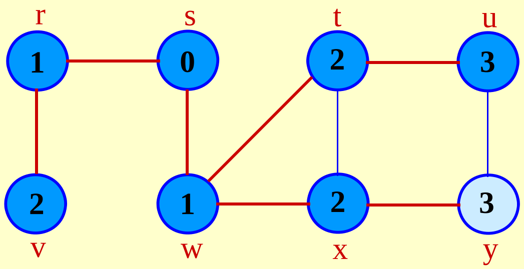
Q: v u y
2 3 3

Example (BFS)



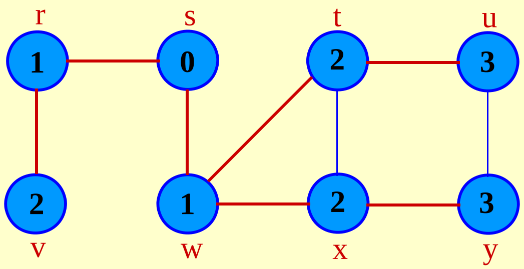
Q: u y
3 3

Example (BFS)



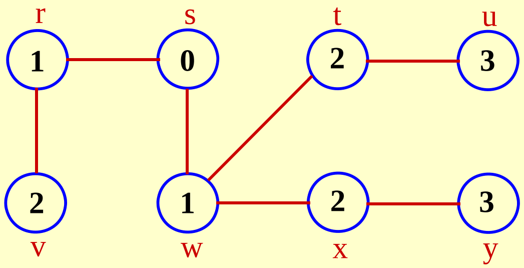
Q: y
3

Example (BFS)



Q: ∅

Example (BFS)



BF Tree

Analysis of BFS

- Initialization takes $O(|V|)$.
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(|V|)$.
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(|E|)$.
- Summing up over all vertices => total running time of BFS is $O(|V| + |E|)$, linear in the size of the adjacency list representation of graph.

Breadth-first Tree

- For a graph $G = (V, E)$ with source s, the **predecessor subgraph** of G is $G_\pi = (V_\pi, E_\pi)$ where
  - $V_\pi=\{v\in V : \pi[v] \neq nil\} \cup \{s\}$
  - $E_\pi=\{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- The predecessor subgraph G_π is a **breadth-first tree** if:
  - V_π consists of the vertices reachable from s and
  - for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G.
- The edges in E_π are called **tree edges**. $|E_\pi| = |V_\pi| - 1$.

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex v.
- When all edges of v have been explored, backtrack to explore other edges leaving the vertex from which v was discovered (its *predecessor*).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - **2 timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
- Coloring scheme for vertices as BFS. A vertex is
  - "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if it is a leaf node or all vertices adjacent to it have been finished.

Pseudo-code

DFS(*G*)

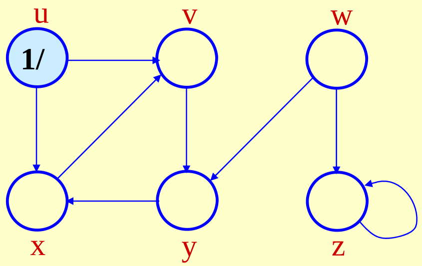
```
for each vertex $u \in V[G]$
    do $color[u] \leftarrow$ white
       $\pi[u] \leftarrow$ NIL
$time \leftarrow 0$
for each vertex $u \in V[G]$
    do if $color[u]$ = white
        then DFS-Visit($u$)
```

Uses a global timestamp ***time***.

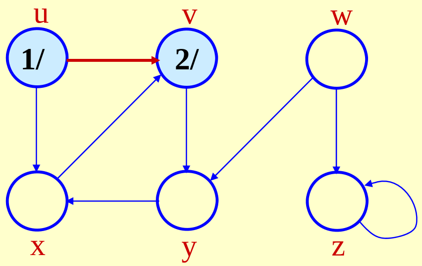
DFS-Visit(*u*)

```
$color[u] \leftarrow$ GRAY   // White vertex $u$ has been discovered
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for each $v \in Adj[u]$
    do if $color[v]$ = WHITE
        then $\pi[v] \leftarrow u$
             DFS-Visit($v$)
$color[u] \leftarrow$ BLACK   // Blacken $u$; it is finished.
$f[u] \leftarrow time \leftarrow time + 1$
```

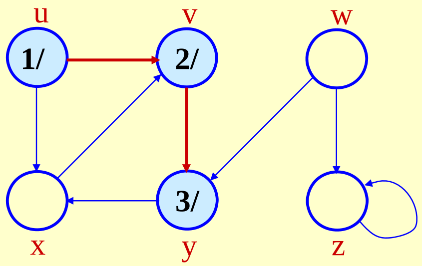
Example (DFS)



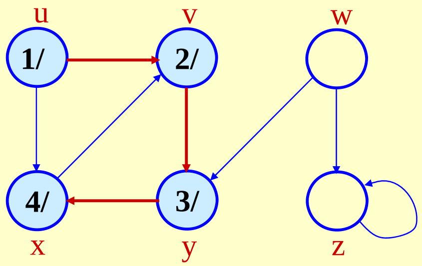
Example (DFS)



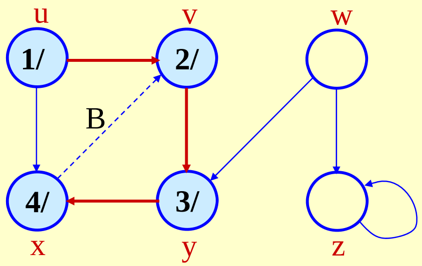
Example (DFS)



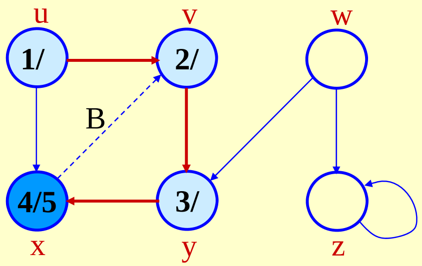
Example (DFS)



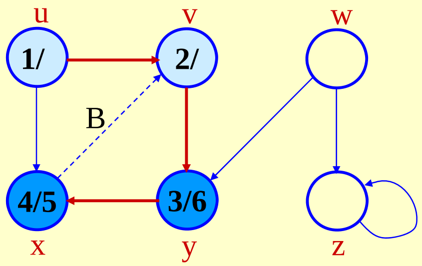
Example (DFS)



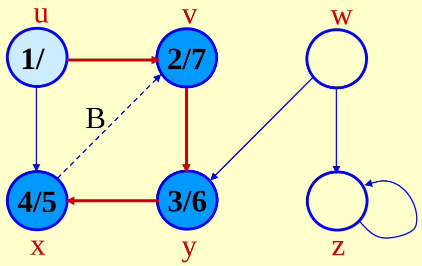
Example (DFS)



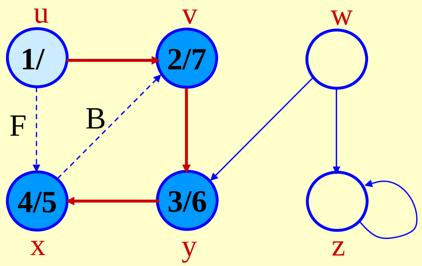
Example (DFS)



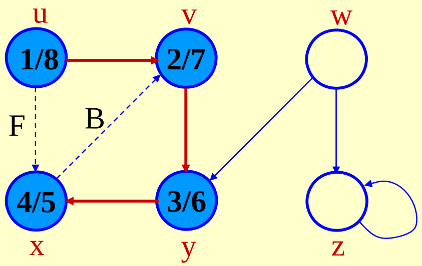
Example (DFS)



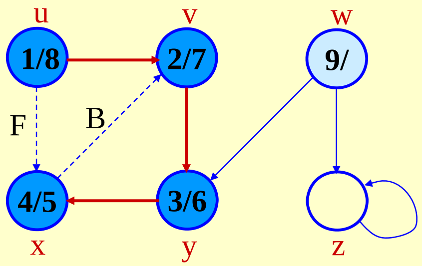
Example (DFS)



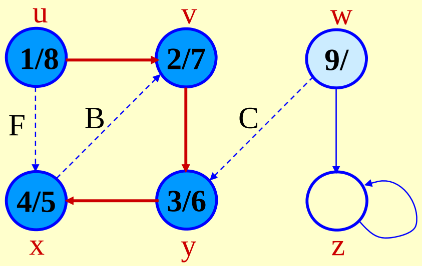
Example (DFS)



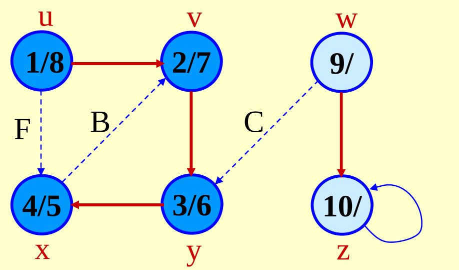
Example (DFS)



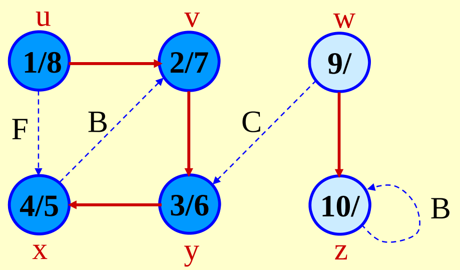
Example (DFS)



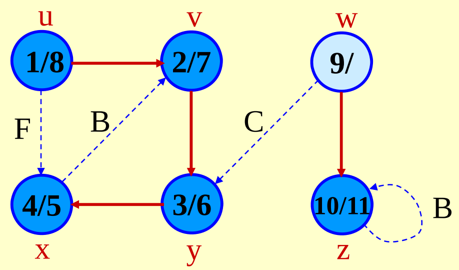
Example (DFS)



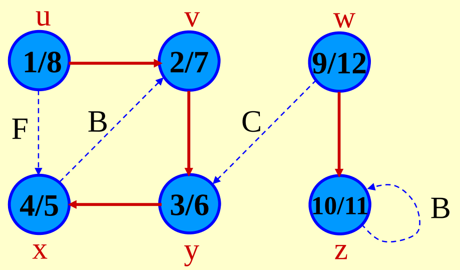
Example (DFS)



Example (DFS)



Example (DFS)



Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed $|\mathrm{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\mathrm{Adj}[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(|V| + |E|)$.

Recursive DFS Algorithm

```
Traverse( )
    for all nodes X
        set X.tag = False
    Visit ( 1st node )
Visit ( X )
    set X.tag = True
    for each successor Y of X
        if (Y.tag = False)
            Visit ( Y )
```

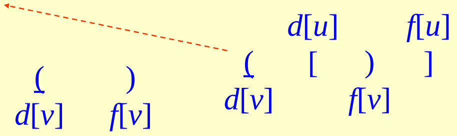
Parenthesis Theorem

Theorem 22.7

For all u, v, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

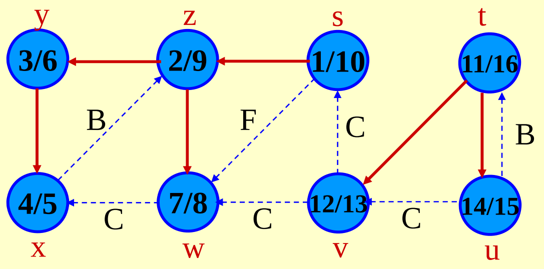
- So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.
- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])



Corollary

v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.

Example (Parenthesis Theorem)



(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi=\{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq nil\}$.
  - » How does it differ from that of BFS?
  - » The predecessor subgraph G_π forms a *depth-first forest* composed of several *depth-first trees*. The edges in E_π are called *tree edges*.

Definition:
Forest: An acyclic graph G that may be disconnected.

White-path Theorem

Theorem 22.9

v is a descendant of *u* in *DF-tree* if and only if at time $d[u]$, there is a path $u \rightsquigarrow v$ consisting of only white vertices. (Except for *u*, which was *just* colored gray.)

Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring (u, v).
- **Back edge:** (u, v), where *u* is a descendant of *v* (in the depth-first tree).
- **Forward edge:** (u, v), where *v* is a descendant of *u*, but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Classifying edges of a digraph

- (u, v) is:
  - » Tree edge – if v is white
  - » Back edge – if v is gray
  - » Forward or cross - if v is black
- (u, v) is:
  - » Forward edge – if v is black and d[u] < d[v] (v was discovered after u)
  - » Cross edge – if v is black and d[u] > d[v] (u discovered after v)

More applications

- **Does directed G contain a directed cycle?** Do DFS if back edges yes. Time O(V+E).
- **Does undirected G contain a cycle?** Same as directed but be careful not to consider (u,v) and (v, u) a cycle.

  Time O(V) since encounter at most |V| edges (if (u, v) and (v, u) are counted as one edge), before cycle is found.
- **Is undirected G a tree?** Do dfsVisit(v). If all vertices are reached and no back edges G is a tree. O(V)

C# Interfaces

```
using System;
using System.Collections.Generic;
using System.Security.Permissions;
[assembly: CLSCompliant(true)]
namespace OhioState.Collections.Graph {
    /// <summary>
    /// IEdge provides a standard interface to specify an edge and any
    /// data associated with an edge within a graph.
    /// </summary>
    /// <typeparam name="N">The type of the nodes in the
        graph.</typeparam>
    /// <typeparam name="E">The type of the data on an edge.</typeparam>
    public interface IEdge<N,E>     {
       /// <summary>
       /// Get the Node label that this edge emanates from.
       /// </summary>
       N From { get; }
       /// <summary>
       /// Get the Node label that this edge terminates at.
       /// </summary>
       N To { get; }
       /// <summary>
       /// Get the edge label for this edge.
       /// </summary>
       E Value { get; }
    }
```

```
/// <summary>
/// The Graph interface
/// </summary>
/// <typeparam name="N">The type associated at each node. Called a
    node or node label</typeparam>
/// <typeparam name="E">The type associated at each edge. Also called
    the edge label.</typeparam>
public interface IGraph<N,E>     {
   /// <summary>
   /// Iterator for the nodes in the graoh.
   /// </summary>
   IEnumerable<N> Nodes { get; }
   /// <summary>
   /// Iterator for the children or neighbors of the specified node.
   /// </summary>
   /// <param name="node">The node.</param>
   /// <returns>An enumerator of nodes.</returns>
   IEnumerable<N> Neighbors(N node);
  /// <summary>
   /// Iterator over the parents or immediate ancestors of a node.
   /// </summary>
   /// <remarks>May not be supported by all graphs.</remarks>
   /// <param name="node">The node.</param>
   /// <returns>An enumerator of nodes.</returns>
   IEnumerable<N> Parents(N node);
```

C# Interfaces

```
/// <summary>
/// Iterator over the emanating edges from a node.
/// </summary>
/// <param name="node">The node.</param>
/// <returns>An enumerator of nodes.</returns>
IEnumerable<IEdge<N, E>> OutEdges(N node);
/// <summary>
/// Iterator over the in-coming edges of a node.
/// </summary>
/// <remarks>May not be supported by all graphs.</remarks>
/// <param name="node">The node.</param>
/// <returns>An enumerator of edges.</returns>
IEnumerable<IEdge<N, E>> InEdges(N node);
/// <summary>
/// Iterator for the edges in the graph, yielding IEdge's
/// </summary>
IEnumerable<IEdge<N, E>> Edges { get; }
/// <summary>
/// Tests whether an edge exists between two nodes.
/// </summary>
/// <param name="fromNode">The node that the edge emanates
 from.</param>
/// <param name="toNode">The node that the edge terminates
 at.</param>
/// <returns>True if the edge exists in the graph. False
 otherwise.</returns>
bool ContainsEdge(N fromNode, N toNode);
/// <summary>
/// Gets the label on an edge.
/// </summary>
/// <param name="fromNode">The node that the edge emanates
 from.</param>
/// <param name="toNode">The node that the edge terminates
 at.</param>
```

```
        /// <returns>The edge.</returns>
        E GetEdgeLabel(N fromNode, N toNode);
        /// <summary>
        /// Exception safe routine to get the label on an edge.
        /// </summary>
        /// <param name="fromNode">The node that the edge emanates
         from.</param>
        /// <param name="toNode">The node that the edge terminates
         at.</param>
        /// <param name="edge">The resulting edge if the method was
         successful. A default
        /// value for the type if the edge could not be found.</param>
        /// <returns>True if the edge was found. False otherwise.</returns>
        bool TryGetEdge(N fromNode, N toNode, out E edge);
    }
}
```

C# Interfaces

```
using System;
  namespace OhioState.Collections.Graph {
    /// <summary>
    /// Graph interface for graphs with finite size.
    /// </summary>
    /// <typeparam name="N">The type associated at each node. Called a node or node
      label</typeparam>
    /// <typeparam name="E">The type associated at each edge. Also called the edge
      label.</typeparam>
    /// <seealso cref="IGraph{N, E}"/>
    public interface IFiniteGraph<N, E> : IGraph<N, E>     {
        /// <summary>
        /// Get the number of edges in the graph.
        /// </summary>
        int NumberOfEdges { get; }
        /// <summary>
        /// Get the number of nodes in the graph.
        /// </summary>
        int NumberOfNodes { get; }
    }
 }
```