

Lecture 10: Graphs, BFS, DFS, and Topological Sort

Course Learning Outcome

- Use fundamental **graph algorithms**, like **traversal**, shortest path and spanning tree in the solution of real-life problems

Dr. Khalil Yousef

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

Read Chapter 22 of *Introduction to Algorithms*

Graphs

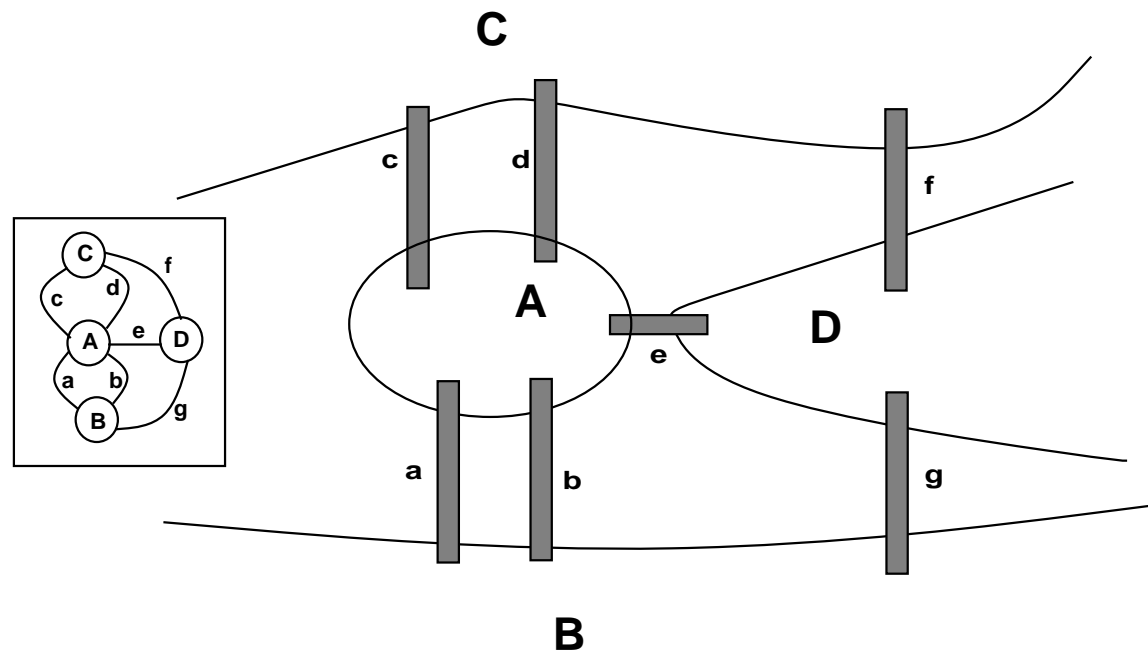
Graph representations of data structures are very useful in a variety of disciplines. Examples of graph applications:

- Network Analysis
- Project Planning (PERT charts)
- Shortest Route Planning
- Compilers: Data-Dependency Graphs, Control-Flow Graphs
- VLSI
- Natural Language Processing

We will discuss techniques for representing graphs and perform some basic operations on them (e.g., breadth-first search, depth-first search, topological sort).

The First Use of Graphs

Graphs were first used by Euler in 1736, when he worked on the **Königsberg Bridge Problem**. There are four land areas (or vertices): A, B, C, D and seven bridges (or edges): a, b, c, d, e, f, g . The problem is to determine whether there is a way to start out from one land area and walk across each of the bridges exactly once and return to the original land area (i.e., Is there a Eulerian walk?).



Definition of a Graph

A graph G consists of a finite, non-empty vertex (or node) set V and an edge set E (possibly empty), i.e., $G = (V, E)$. Note that $E \subseteq V \times V$; hence $|E| = O(V^2)$.

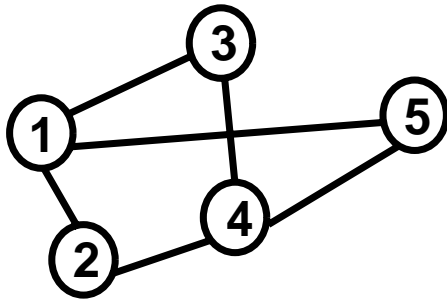
If $u, v \in V$ and there is an edge between those vertices, we can represent this by storing an ordered pair (u, v) in E . Normally, (u, v) appears in E only once (unless you are working with a **multigraph**, which allows multiple edges between vertices).

If an edge between two vertices is **directed** from one vertex to the other, $u \rightarrow v$, it is represented as a tuple $(u, v) \in E$. Note that (v, v) represents a self-looping directed edge.

If an edge between two vertices is **undirected**, then for $(i, j) \in E$, $(i, j) = (j, i)$. There are no self-loops allowed, i.e., $(v, v) \notin E$.

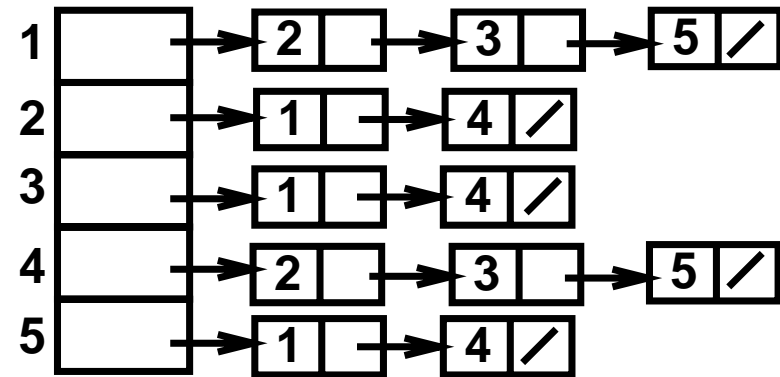
Example of an Undirected Graph

Undirected Graph, G



$G = (V, E)$
 $V = \{1, 2, 3, 4, 5\}$
 $E = \{(1,2), (1,3), (1,5),$
 $\quad (2,4), (3,4), (4,5)\}$

Adjacency List for G

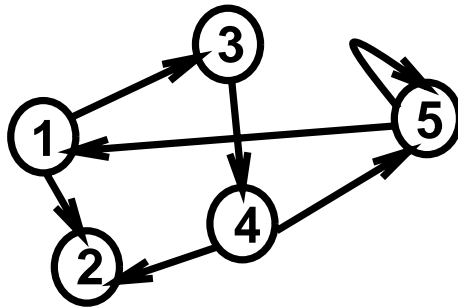


Adjacency Matrix for G

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	0	1	0
3	1	0	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0

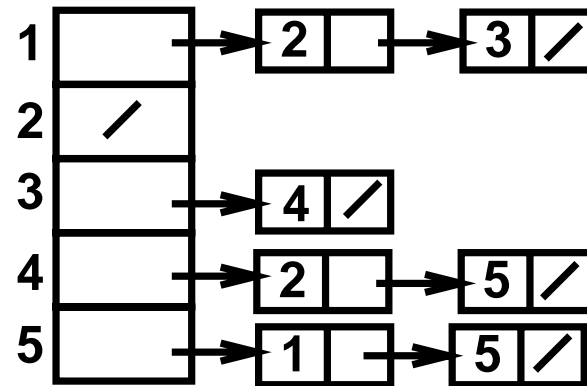
Example of a Directed Graph (or Digraph)

Directed Graph, G



$G = (V, E)$
 $V = \{1, 2, 3, 4, 5\}$
 $E = \{(1,2), (1,3), (3,4), (4,2),$
 $(4,5), (5,1), (5,5)\}$

Adjacency List for G



Adjacency Matrix for G

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	0
3	0	0	0	1	0
4	0	1	0	0	1
5	1	0	0	0	1

1. Graph Representation: Adjacency Lists

For graph $G = (V, E)$, an adjacency list representation consists of an array of length $|V|$, Adj , such that for each vertex $v \in V$, $Adj[v]$ keeps a list of those vertices adjacent to v .

Given that the **degree** of a vertex v is the number of incident edges to v in an undirected graph $G = (V, E)$, we know that the number of items in an adjacency list representing G is $\sum_{v \in V} degree(v) = 2|E|$. Hence, the amount of storage required to represent G is $\Theta(V + E) = \Theta(max(V, E))$.

A directed graph's vertices have both **in-degree** and **out-degree**. The number of items stored in an adjacency list for a directed graph $G = (V, E)$ is $\sum_{v \in V} out-degree(v) = |E|$, resulting in storage requirements of $\Theta(V + E) = \Theta(max(V, E))$.

Adjacency lists are good representations for sparse graphs $|E| \ll |V|^2$; however, there is no quick way to determine whether a given edge (u, v) is present in the graph without searching the list associated with $Adj[u]$.

2. Graph Representation: Adjacency Matrix

For graph $G = (V, E)$, an adjacency matrix consists of a $|V| \times |V|$ matrix A , such that:

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

Using this method, we can determine very quickly whether there is an edge between two vertices; however, it uses $\Theta(V^2)$ memory, independent of the size of E .

Note that for an undirected graph G represented with an adjacency matrix A that $A = A^T$, i.e., $A[i, j] = A[j, i]$, which can be used to cut the memory requirements in half. We can further reduce space by using a bit matrix representation.

This is a good representation for dense graphs, i.e., $|E| \approx |V|^2$ or for small graphs. Usually this method requires too much storage for large graphs, especially since many graphs are sparse (e.g., planar graphs).

Weighted Graphs

In a **weighted graph** G , weights can be associated with the edges or the vertices of the graph.

If each edge has an associated **weight** given by a weight function $w : E \rightarrow \mathbf{R}$, adjacency-list and adjacency-matrix representations must be modified to represent this additional information.

The adjacency list for G can be easily adapted to represent this weight information. For each edge $(u, v) \in E$, store the weight $w(u, v)$ with vertex v in u 's adjacency list.

Adjacency matrices can also be easily adapted to represent a weighted graph, G . For each edge $(u, v) \in E$, store the weight $w(u, v)$ in $A[u, v]$ (NIL if no edge). Note that no bit-matrix representation is possible in this case.

Breadth-First Search

Given a directed or undirected graph $G = (V, E)$ and a distinguished source vertex s , the breadth-first search algorithm systematically explores the edges of G to discover whether every vertex is reachable from s . It does this by visiting all vertices at distance k before vertices at distance $k + 1$. The algorithm:

- computes the distance (fewest number of edges) from the source vertex to every other vertex,
- creates a breadth-first tree with root s to all reachable vertices such that the path from s to v represents the shortest path between those two nodes.

Ancestors and descendants in the breadth-first tree are defined with respect to the shortest path from the source to each vertex.

Breadth-First Search Algorithm: BFS

BFS:

- assumes that $G = (V, E)$ is represented using an adjacency list.
- uses a first-in-first-out (FIFO) queue, Q , to manage the traversal of vertices.
- keeps track of the color of a vertex $u \in V$ using $color[u]$. If u is WHITE, it has not been discovered; if GRAY, it has been put on Q ; if BLACK, its successors have been added to Q .
- keeps track of the distance between the source vertex s and $u \in V$ in $d[u]$.
- uses $\pi[u]$ to store the predecessor of u in order to construct a BFS-Tree. If there is no predecessor, the value of $\pi[u]$ is NIL.

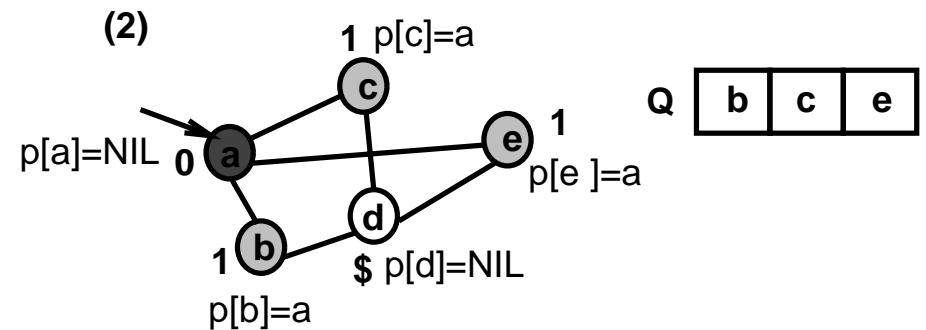
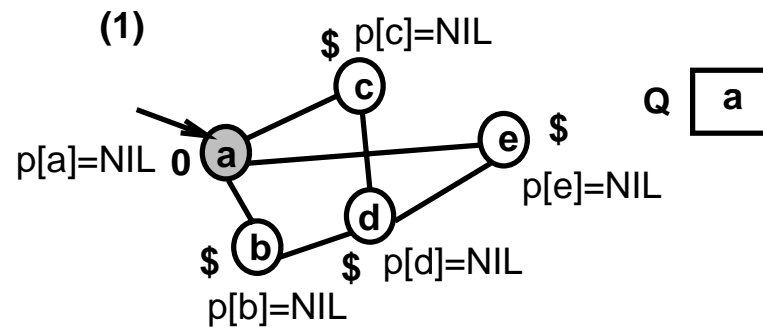
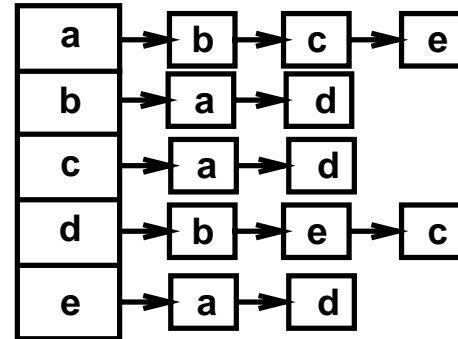
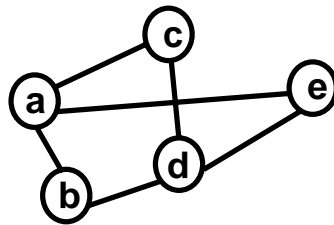
BFS(G, s)

1. **for** each vertex $u \in V[G] - \{s\}$
2. **do** $color[u] \leftarrow \text{WHITE}$
3. $d[u] \leftarrow \infty$
4. $\pi[u] \leftarrow \text{NIL}$

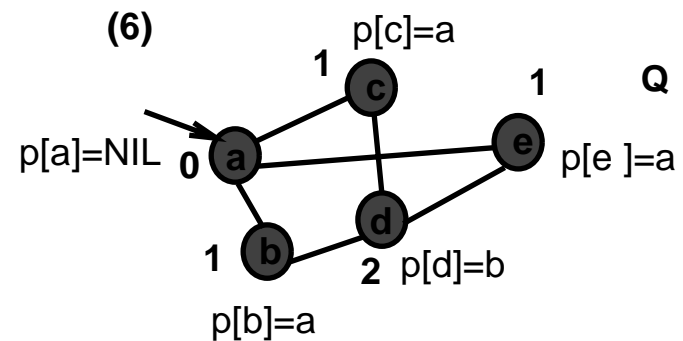
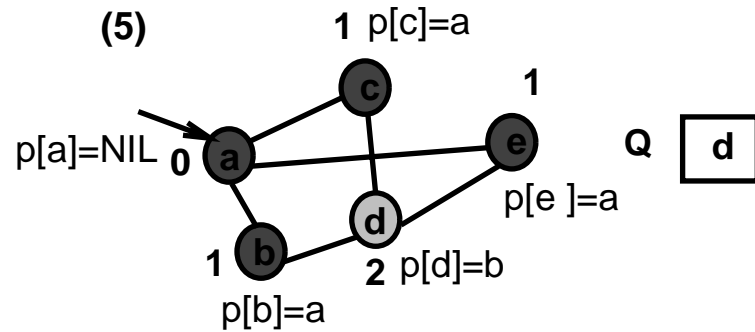
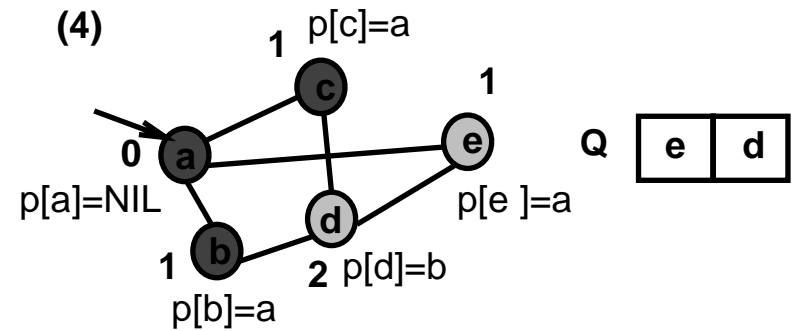
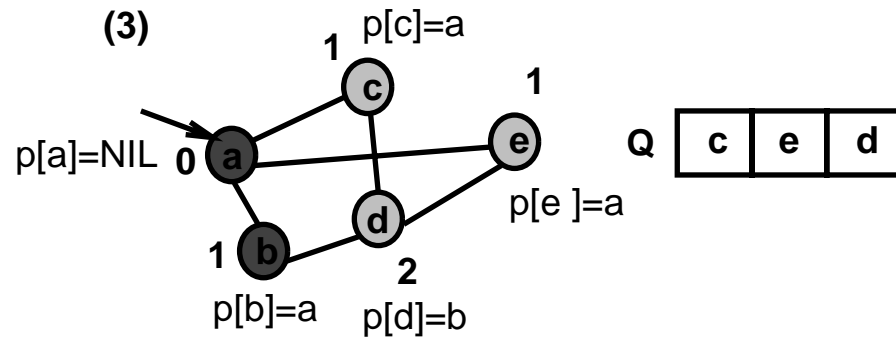
Breadth-First Search Algorithm: BFS *continued*

```
5.  $color[s] \leftarrow \text{GRAY}$ 
6.  $d[s] \leftarrow 0$ 
7.  $\pi[s] \leftarrow \text{NIL}$ 
8.  $Q \leftarrow \{s\}$ 
9. while  $Q \neq \emptyset$ 
10.     do  $u \leftarrow head[Q]$ 
11.         for each  $v \in Adj[u]$ 
12.             do if  $color[v] = \text{WHITE}$ 
13.                 then  $color[v] \leftarrow \text{GRAY}$ 
14.                      $d[v] \leftarrow d[u] + 1$ 
15.                      $\pi[v] \leftarrow u$ 
16.                      $\text{ENQUEUE}(Q, v)$ 
17.      $\text{DEQUEUE}(Q)$ 
18.      $color[u] \leftarrow \text{BLACK}$ 
```

Example: BFS(G, a)



Example: $\text{BFS}(G, a)$ *continued*



Complexity of BFS

- **Initialize:** $|V| - 1$ vertices are initialized to WHITE in $\Theta(V)$.
- **Queue Operations:** Because vertices when enqueued become GRAY and they are never reset to WHITE, vertices are enqueued at most once. Each enqueue and dequeue operation uses $O(1)$ time; hence, all queue operations take $O(V)$ time.
- **Scanning Adjacency Lists:** The lists are scanned at most once, right after dequeuing. The sum of their lengths is $\Theta(E)$. Hence, the time to scan the lists is $O(E)$.
- **Resulting Complexity:** $O(V + E)$

The Depth-First Search Algorithm: DFS

DFS searches deeper in a graph $G = (V, E)$ before completing the exploration of the vertices on a certain level. This is done by exploring all edges out of the most recently discovered vertex v before backtracking to explore v 's sibling vertices.

In DFS, there does not need to be a distinct source vertex from which to start. The resulting predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a **depth-first forest** with the following properties:

- All vertices are included in the resulting forest, including disconnected nodes.
- $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{nil}\}$ is the set of **tree edges**.

The Depth-First Search Algorithm: DFS *continued*

DFS:

- assumes that $G = (V, E)$ is represented using an adjacency list.
- uses backtracking to manage the traversal of vertices.
- keeps track of the color of a vertex $u \in V$ using $color[u]$. If u is WHITE, it has not been discovered; if GRAY, it has been discovered; if BLACK, its adjacency list has been completely explored.
- uses $\pi[u]$ to store the predecessor of u in order to construct a DFS-Forest. If there is no predecessor, the value of $\pi[u]$ is NIL.
- keeps track of two timestamps for each vertex $v \in V$; $d[v]$ records when v is first discovered and grayed; $f[v]$ records when v 's adjacency list has been completely examined, at which point v is made BLACK. Note that $d[v] < f[v]$ and both timestamps are integers in the range $[1, 2|V|]$. Each time stamp will be set only one time for each vertex v .

DFS(G)

DFS(G)

1. **for** each vertex $u \in V[G]$
2. **do** $color[u] \leftarrow \text{WHITE}$
3. $\pi[u] \leftarrow \text{NIL}$
4. $time \leftarrow 0$ \triangleright global timestamp
5. **for** each vertex $u \in V[G]$
6. **do if** $color[u] = \text{WHITE}$
7. **then** DFS-VISIT(u)

DFS-VISIT(u)

DFS-VISIT(u)

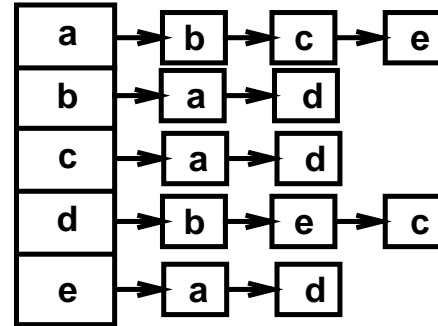
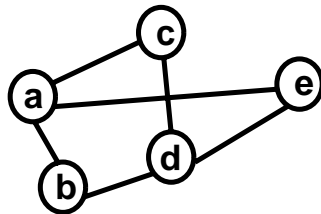
1. $color[u] \leftarrow \text{GRAY}$
2. $d[u] \leftarrow time \leftarrow time + 1$
3. **for** each vertex $v \in Adj[u]$
4. **do if** $color[v] = \text{WHITE}$
5. **then** $\pi[v] \leftarrow u$
6. DFS-VISIT(v)
7. $color[u] \leftarrow \text{BLACK}$
8. $f[u] \leftarrow time \leftarrow time + 1$

Complexity of DFS

- **Initialize:** $|V|$ vertices are initialized to WHITE in $\Theta(V)$.
- **DFS-VISIT:** is called once for each vertex $v \in V$ because vertices are never reset to WHITE. During the execution of $\text{DFS-VISIT}(v)$, the loop at lines 3-6 is executed $|Adj[v]|$ times. Because $\sum_{v \in V} |Adj[v]| = \Theta(E)$, the total cost to execute the loop over all vertices is $\Theta(E)$.
- **Resulting Complexity:** $O(V + E)$

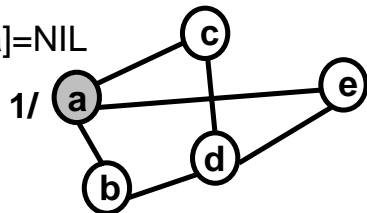
Example: DFS(G)

$V=\{a,b,c,d,e\}$

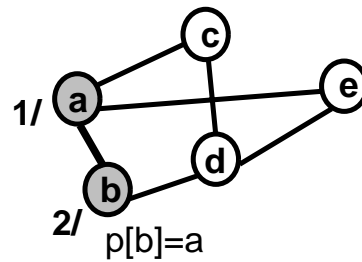


(1)

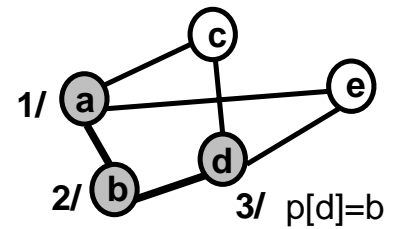
$p[a]=NIL$



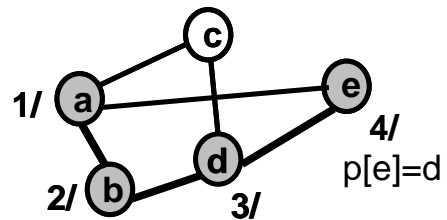
(2)



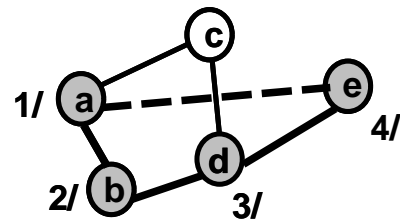
(3)



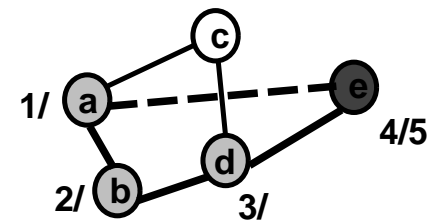
(4)



(5)

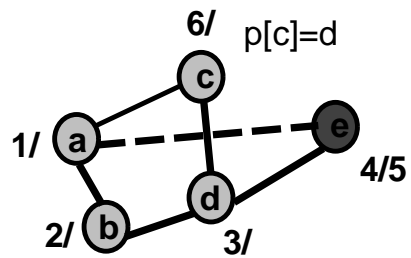


(6)

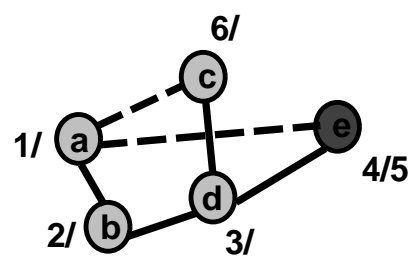


Example: DFS(G) *continued*

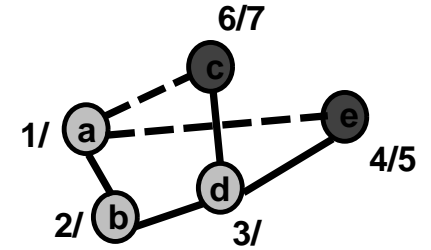
(7)



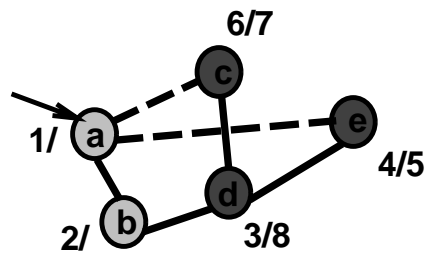
(8)



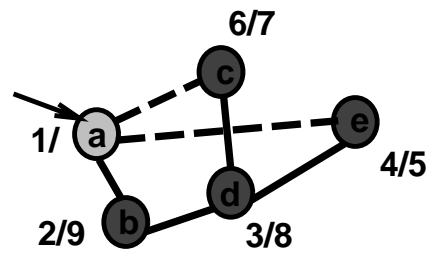
(9)



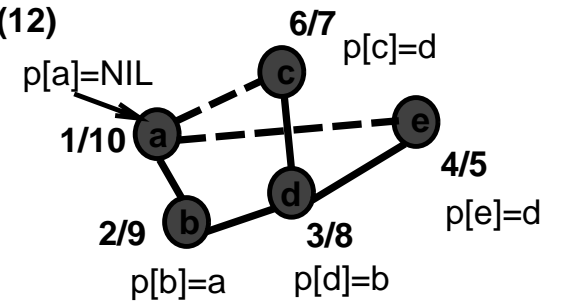
(10)



(11)



(12)



DFS(G) Edge Classification

The edges in directed G can be classified as follows by DFS(G):

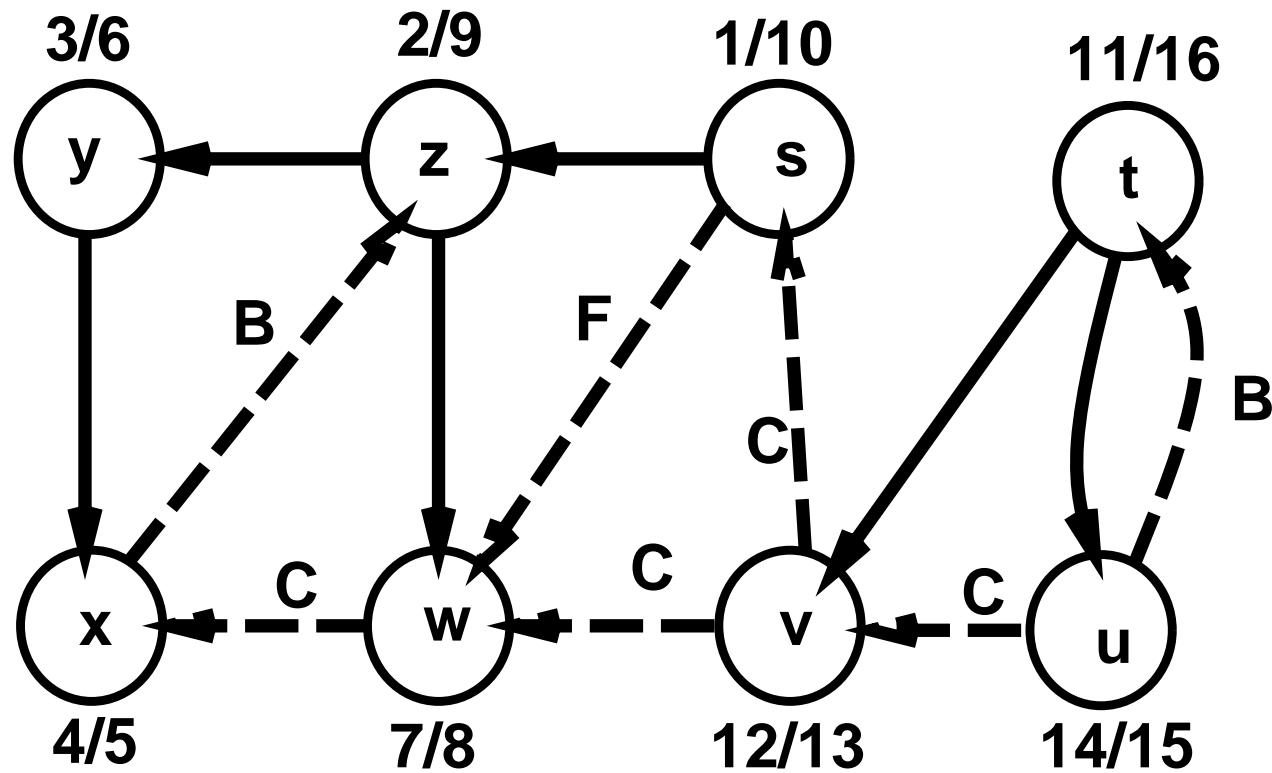
1. **Tree Edge:** an edge (u, v) in a DFS-Forest G_π resulting from the discovery of a vertex v from a GRAY vertex u ($d[u] < d[v] < f[v] < f[u]$). When the edge (u, v) is first explored v is WHITE.
2. **Back Edge:** an edge (u, v) connecting vertex u to an ancestor v (or u itself) ($d[v] < d[u] < f[u] < f[v]$). When the edge (u, v) is first explored v is GRAY.
3. **Forward Edge:** a non-tree edge (u, v) connecting u to a descendent v in a DFS-Tree ($d[u] < d[v] < f[v] < f[u]$). When the edge (u, v) is first explored v is BLACK.
4. **Cross Edge:** all remaining edges; either within a single tree where there is no ancestor relationship between the vertices or edges between two different trees in the forest ($d[v] < f[v] < d[u] < f[u]$). When the edge (u, v) is first explored v is BLACK.

DFS(G) Edge Classification

More about the Forward and Cross Edges.

1. **Forward Edge:** Forward edges are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree. Forward edges describe **ancestor-to-descendant relations**, as they lead from **low to high nodes**.
2. **Cross Edge:** Cross edges are all other edges. They can go between vertices in the same depth-first tree as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees. Cross edges link nodes with **no ancestor-descendant relation** and point from **high to low nodes**.

DFS(G) Edge Classification Example



$V = \{s, z, y, x, w, t, v, u\}$

Topological Sort

A **topological sort** of a directed acyclic graph (or DAG), $G = (V, E)$ is a linear ordering of all of its vertices such that if $(u, v) \in E$, then u appears before v in the ordering.

A DAG expresses a partial order on its vertices; a topological sort generates a linear ordering of the vertices such that the partial order relations are preserved.

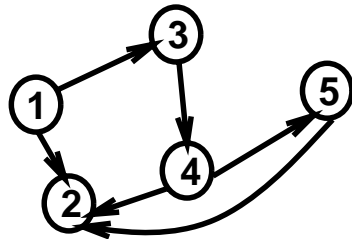
Basic Idea: The finish times produced by DFS form a total order needed to produce a topological sort of a DAG.

TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute finish times $f[v]$ for each $v \in V$.
2. as each vertex is finished, insert it into the front of a linked list.
3. **return** the linked list of vertices.

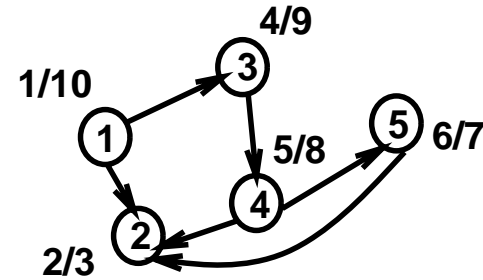
TOPOLOGICAL-SORT Example

Directed Acyclic Graph, G

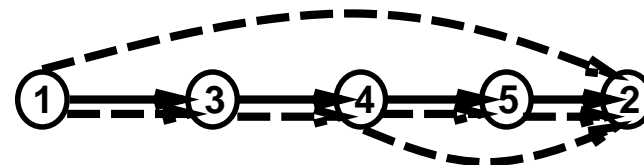


$G = (V, E)$
 $V = \{1, 2, 3, 4, 5\}$
 $E = \{(1,2), (1,3), (3,4), (4,2), (4,5), (5,2)\}$

Labels after DFS



Topological Sort



E 
Topological Sort 

The time to perform TOPOLOGICAL-SORT is $\Theta(V + E)$ because DFS takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of a linked list.