CSC 317: Data Structures and Algorithm Analysis

Dilip Sarkar

Department of Computer Science
University of Miami

Outline I

- Adjacency-Matrix Representation of Graphs
- Adjacency-List Representation
- Edge-List Representation
- Comparison of Complexities of Operations for Different Graph Representations
- Breadth-first search
- Breadth-first trees
- Overview of the DFS algorithm
- DFS Algorithm
- Complexity analysis of the DFS algorithm
- Properties of the DFS algorithm
- Application of DFS: Topological Sorting or Ordering Algorithm
- Application of DFS: Strongly connected components

Adjacency-Matrix Representation of Undirected Unweighted Graphs

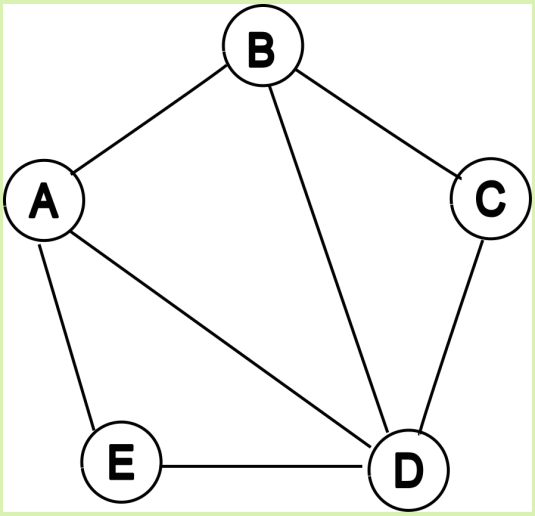


Figure: A simple undirected unweighted graph.

	A	B	C	D	E
A	–	1	0	1	1
B	1	–	1	1	0
C	0	1	–	1	0
D	1	1	1	–	1
E	1	0	0	1	–

Adjacency-matrix representation of simple (no self-loop) undirected unweighted graph.

1: an edge, 0: no edge –: no self-loop

Adjacency-Matrix Representation of Directed Unweighted Graphs

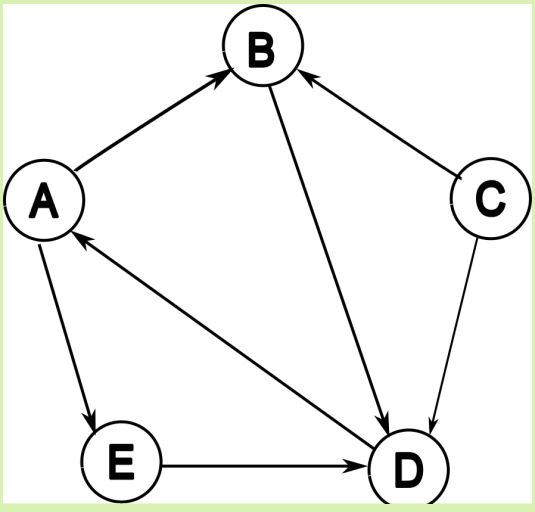


Figure: A Simple Directed Unweighted Graph.

	A	B	C	D	E
A	–	1	0	0	1
B	0	–	0	1	0
C	0	1	–	1	0
D	1	0	0	–	0
E	0	0	0	1	–

Adjacency-matrix representation of simple (no self-loop) directed unweighted graph.

1: an edge, 0: no edge –: no self-loop

Adjacency-Matrix Representation of Directed Weighted Graphs

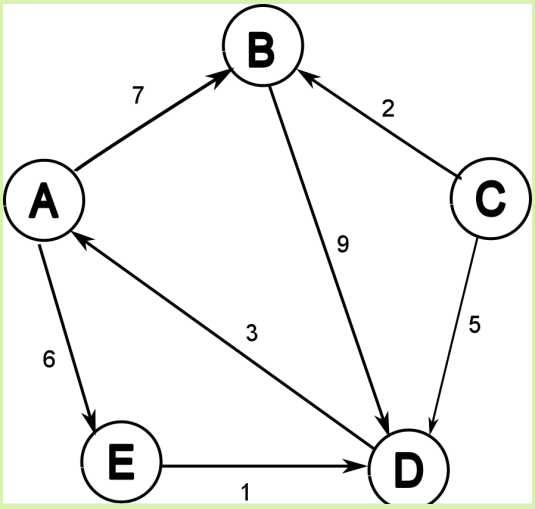


Figure: A Simple Directed Weighted Graph

	A	B	C	D	E
A	–	7	0	0	6
B	0	–	0	9	0
C	0	2	–	5	0
D	3	0	0	–	0
E	0	0	0	1	–

Adjacency-matrix representation of simple (no self-loop) directed unweighted graph.

$w \neq 0$: an edge with weight w, 0: no edge $-$: no self-loop

Adjacency-List Representation of Undirected Unweighted Graphs

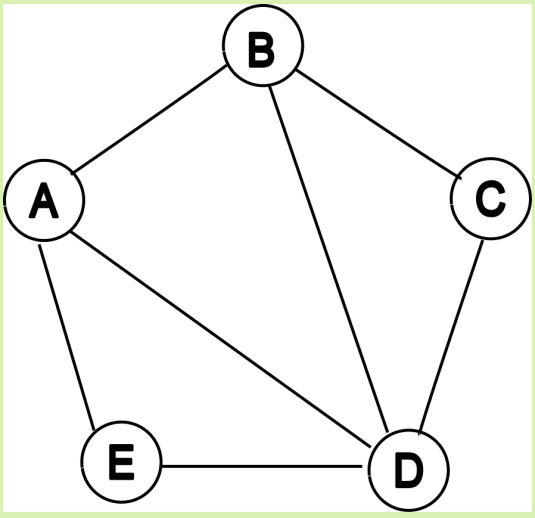


Figure: A simple undirected unweighted graph.

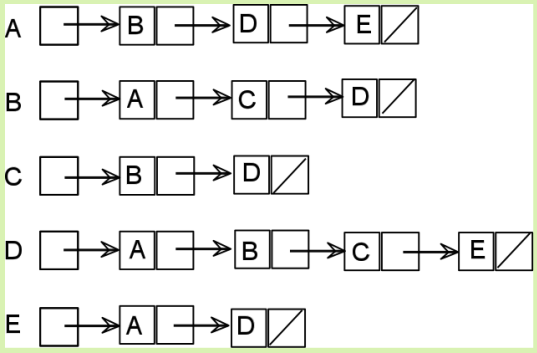


Figure: Adjacency-lis representation of a graph

Adjacency-List Representation of Directed Weighted Graphs

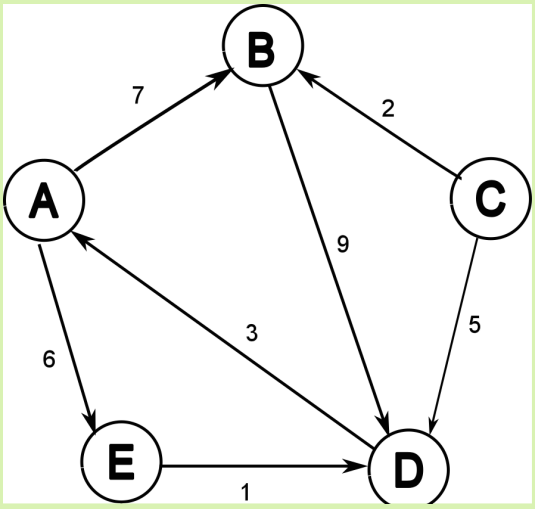


Figure: A Simple Directed Weighted Graph

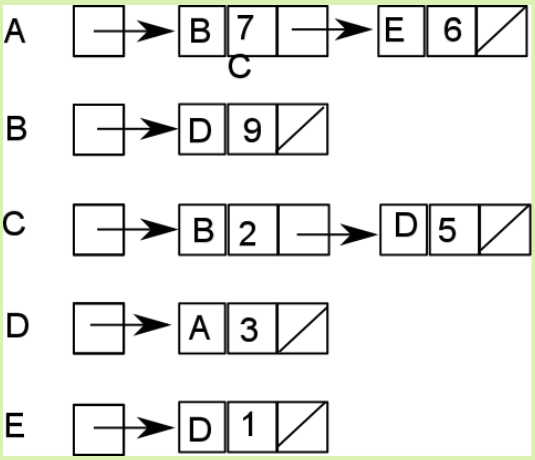


Figure: Adjacency-list representation of directed weighted graph

Edge-List Representation of Undirected Unweighted Graphs

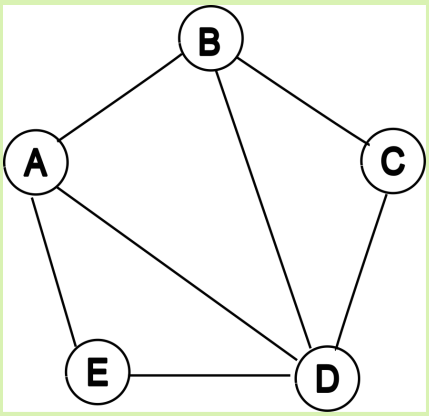


Figure: A simple undirected unweighted graph.

End 1	A	A	A	B	B	C	D
End 2	B	D	E	C	D	D	E
Weight	1	1	1	1	1	1	1

Edge-List Representation of Directed Weighted Graphs

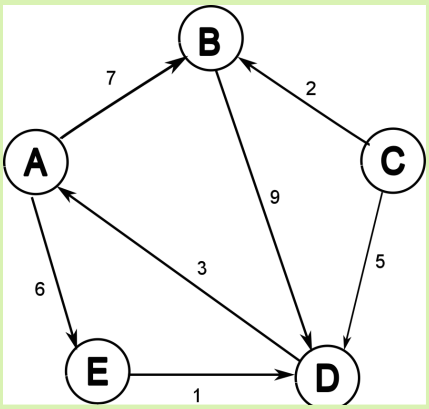


Figure: A Simple Directed Weighted Graph

End 1	A	A	B	C	C	D	E
End 2	B	E	D	B	E	A	D
Weight	7	6	9	2	5	3	1

Complexity Comparison

Operation	Adjacency matrix	Adjacency list	Edge list
Find an edge	$O(1)$	$O(n)$	$O(m)$
Degree of a node (or # of neighbors)	$O(n)$	$O(n)$	$O(m)$
Minimum weight neighbor	$O(n)$	$O(n)$	$O(m)$
Maximum weight neighbor	$O(n)$	$O(n)$	$O(m)$

Table: Comparison of Complexities of Graph Representations

Graph Search Algorithms

- **Notations**: $G(V, E)$ denotes a **graph** with a set of V **vertices or nodes** and E is a set of **edges**.

  $V = \{v_1, v_2, \cdots, c_n\}$ denotes the set of n edges and

  $E = \{e_1, e_2, \cdots, e_m\}$ denotes the set of m edges such that $e_i = (v_{i_1}, v_{i_2})$.
- A graph $G(V, E)$ is a generalization of a tree, because

  A tree with n nodes has exactly $n - 1$ edgers and

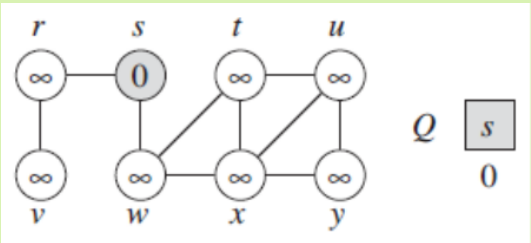
  Between to nodes v_i and v_j, there is only one path.
- A graph with n nodes can have 0 to $\frac{n(n-1)}{2}$ edges.
- **Graph Search:** Two possibilities: Visit every node and visit every edge; a third would be visit every node **and** every edge.
- We will learn two elementary graph search algorithms and their applications:
  - **Breadth-first-search**: Shortest path and minimum spanning tree for unweighted graphs.
  - **Depth-first-search**: Minimal spanning tree, Topological sorting, strongly connected components in a graph, AI search algorithms.

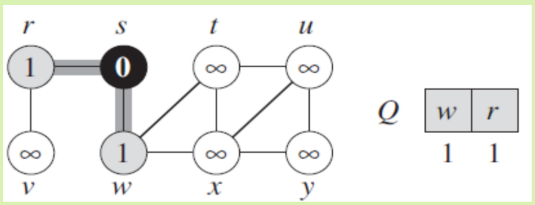
Breadth-first search (BFS) An example – I

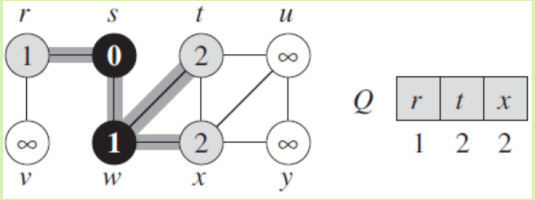
Trace of execution of a BFS algorithms on a graph

- Two data structures are used:
  - Representation of graph adjacency list
  - A FIFO (first-in-first-out) queue, Q, for nodes to be considered next

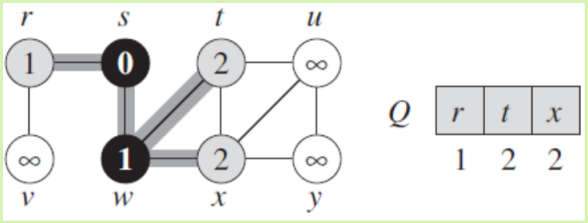
Initialized graph and the Q

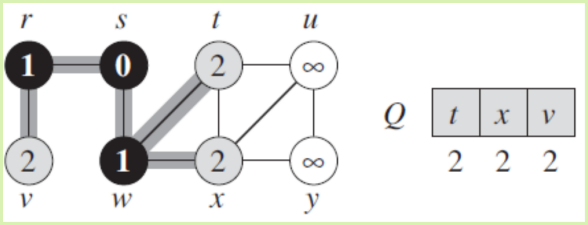


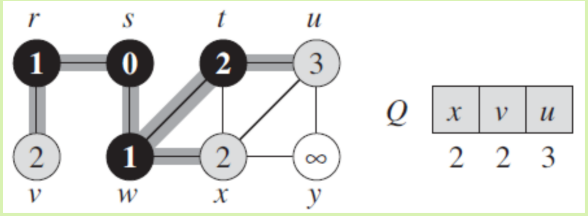


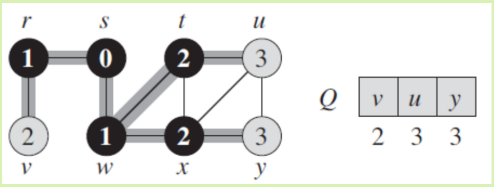


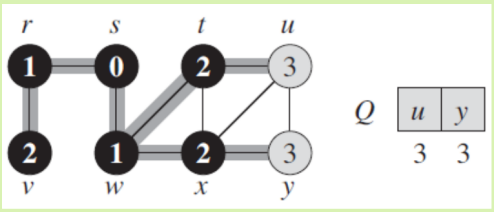
Breadth-first search (BFS): An example – II

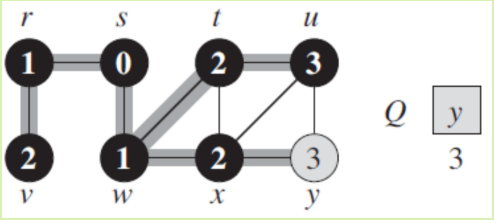


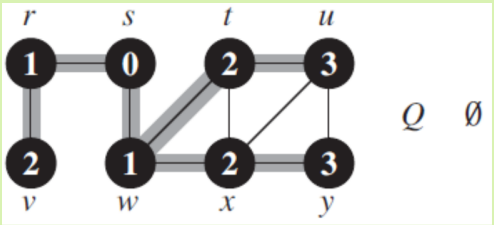












Breadth-first search (BFS) Algorithm

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

- Data structures:
  - Representation of graph adjacency list
  - A FIFO (first-in-first-out) queue, Q for nodes to be considered next

Complexity Analysis I

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

- Initialization: Lines 1 to 4 time complexity $O(n)$, becasue fro loop is executed $(n-1)$ times

  Lines 5 to 9 are executed only once.
- **While** loop is executed $O(n)$ time,

  Because a node is enqueued only once at Line 17
- The **for** loop for each node is executed for each edge

  Each edge is visited twice, once from each node it connects
- Total complexity is thus m, the number of edges.
- **Complexity of BSF** is $O(|V| + |E|) = O(n + m)$

Complexity Analysis II

- **Notations:** Shortest-path distance - number of edges - from s to v is denoted as $\delta(s, v)$

Lemma (Lemma 22.1)

Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Proof.

Outline: Algorithm Lines 11 and 12: we reach v from u. Thus, distance from s to v is at least *one* more than that for u. □

Complexity Analysis III

Lemma (Lemma 22.2)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$ the value of $v.d \geq \delta(s, v)$.

Proof.

Outline: Proof is by induction on the number of ENQUEUE operations. At Line 2 $v.d$ is assigned to inf and at Line 6 $v.d$ is assigned '0'.

Induction hypothesis holds for s, because $s.d = 0 = \delta(s, s)$, and $v.d = \text{inf} \geq \delta(s, v)$ for all $v \in (V - \{s\})$.

- From Line 15 and Lemma 22.1, we have
- $v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$

□

Complexity Analysis IV

Lemma (Lemma 22.3)

Suppose that during the execution of the BFS on a graph $G(V, E)$, the Q contains the vertices $< v_1, v_2, \cdots, v_r >$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \cdots, r - 1$.

Proof.

Outline: Proof is by induction on the number of ENQUEUE operations. □

Complexity Analysis V

Theorem (Theorem 2.5)

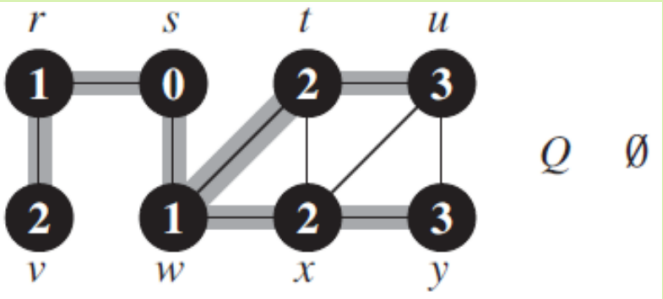
Let $G = (V, E)$ be adirected or undirected graph, and suppose that BFS *is run on G from a given source node $s \in V$. Then, during its execution,* BFS *discovers every vertex $v \in V$ that is reachable from the source node s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$.*
Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Proof.

Outline: Proof is by contradiction. Assume that some vertex receives a d value not equal to its shortest-path distance and show that this contradicts following the steps of the algorithm. □

Breadt-first trees

Corresponding to the π attributes we have tree.



The highlighted edges corresponds the BFS tree.

- Formally, for a given graph $G(V, E)$ with source node s, we define the **predecessor subgraph** of G as $G_\pi(V_\pi, E_\pi)$, where

  $V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \bigcup \{s\}$

  and

  $E_\pi = \{(v.\pi, v) : V.\pi \neq \text{NIL}\}$ - { s } }

- The subgraph G_π is a *breadth-first tree*.

DFS: Depth-first search

- Search **deeper** in the graph, if possible
- Explore edges from the **most recently** discovered vertex v
- Once **all edges** from the vertex v **has been explored**
- **Backtrack** to the vertex u, from where vertex v was discovered
- Repeat DFS from all undiscovered vertices

  This is DIFFERENT from BFS
- For disconnected graphs DFS may produce multiple DFS-trees

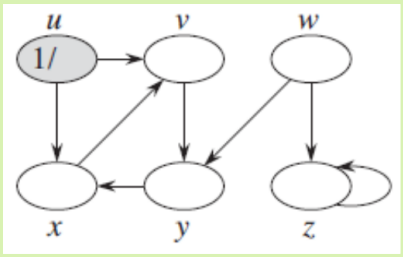
  Called DFS forest
- Every node will be marked twice:

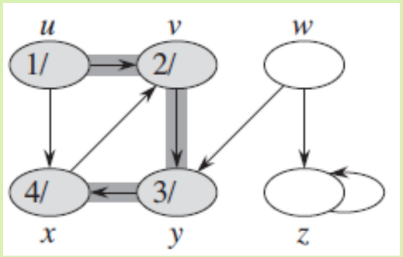
  $v.d$ stores 'time/step' when v is discovered

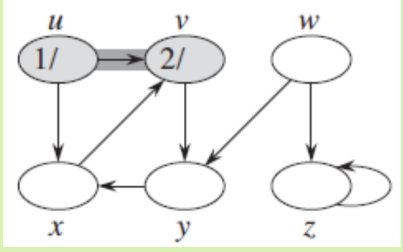
  $v.f$ stores 'time/step' when DFS finish visiting all nodes reachable from v

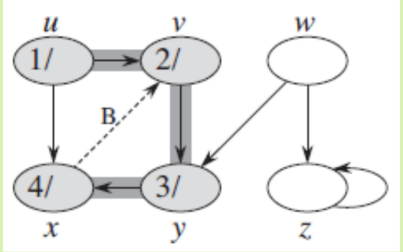
  It is obvious that $v.d < v.f$
- Time stamps at a node stores important information about structure of the graph.

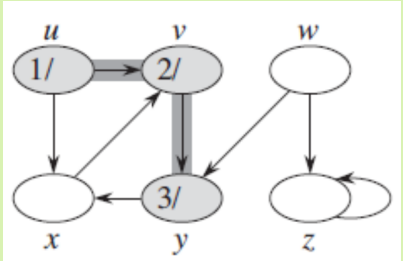
Trace of execution of a DFS on a graph I

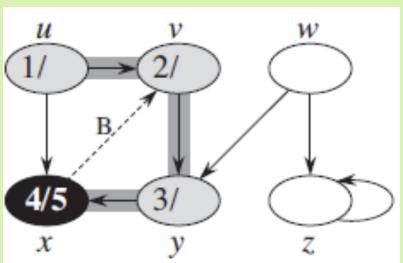




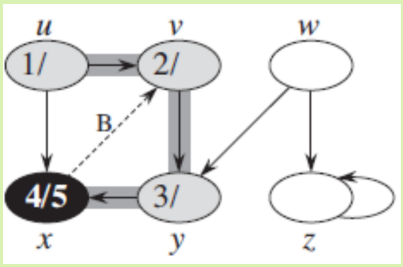


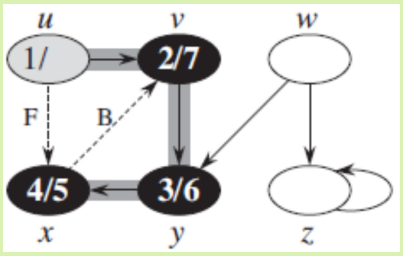


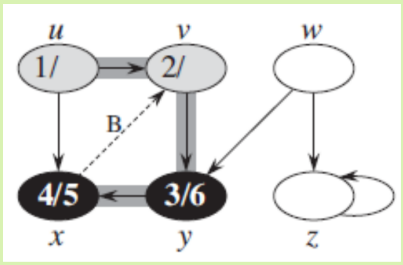


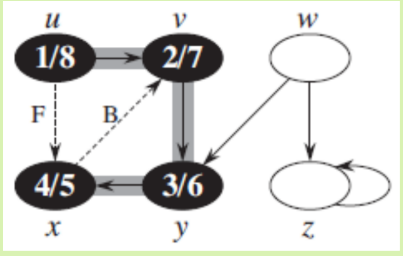


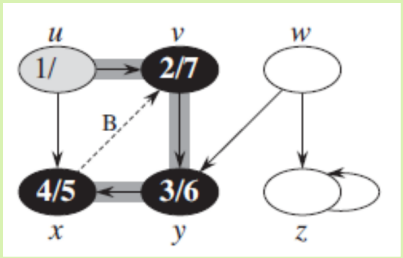
Trace of execution of a DFS on a graph II

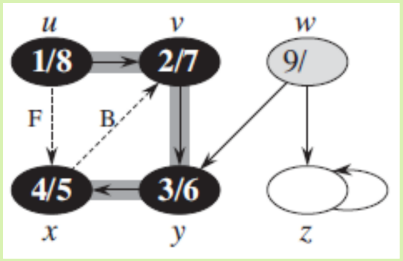




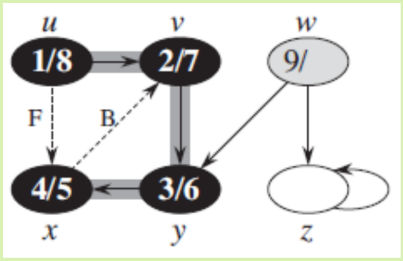


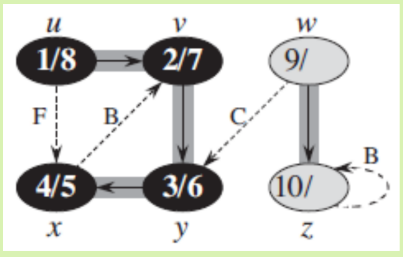


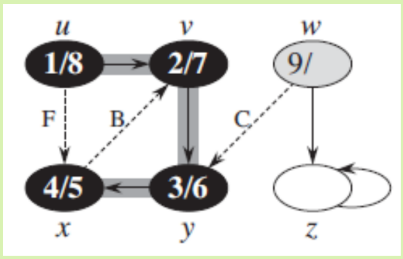


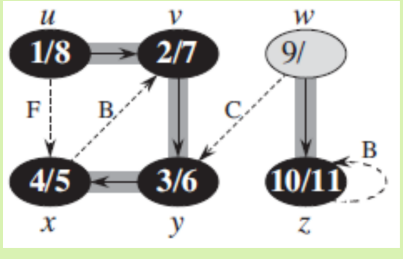


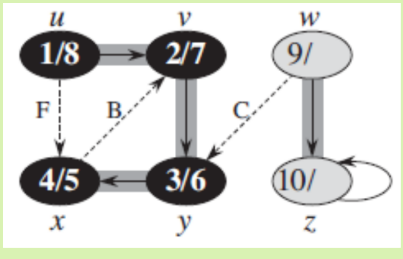
Trace of execution of a DFS on a graph III

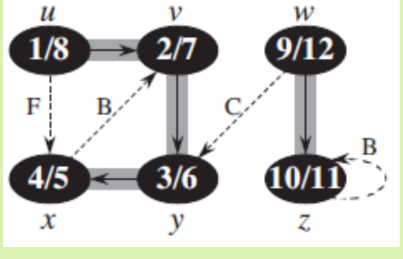












DFS Algorithm

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1   time = time + 1
2   u.d = time
3   u.color = GRAY
4   for each v ∈ G.Adj[u]
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK
9   time = time + 1
10  u.f = time
```


Complexity analysis of the DFS algorithm

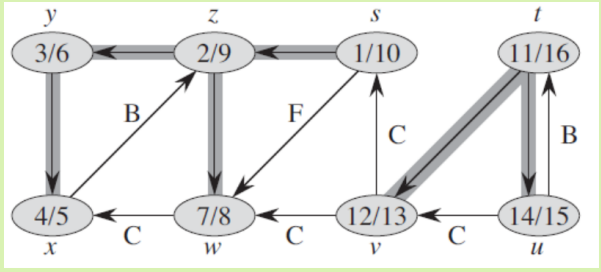
- Initialization Steps 1 to 4 in $\mathrm{DFS}(G)$ takes $O(n)$ time, where n is the number of vertices.
- **For** loop in Steps 5 to 7 in $\mathrm{DFS}(G)$ and Steps 4 to 7 in $\mathrm{DFS}(G, v)$ visits each edge only once.
- Thus, complexity of the DFS algorithm is $O(n + m)$, where m is the number of edges in G.

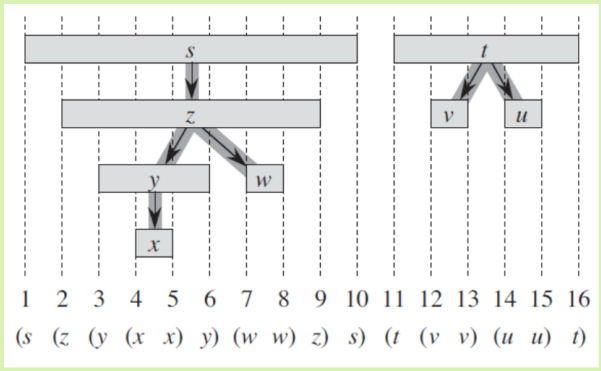
Properties of the DFS algorithm I

Theorem (Parenthesis theorem)

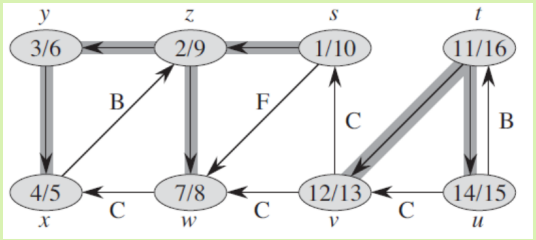
In any depth-first search of a (directed or undirected) graph $G(V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- *the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,*
- *the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in the depth-first tree, or*
- *the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in the depth-first tree.*





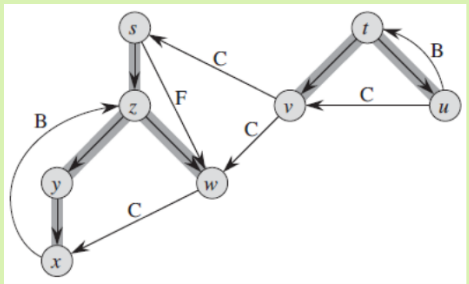
Classification of Edges I



- **Tree Edges** are the edges in the DFS forest.

  Edge (u, v) is a tree edge if v was first *discovered* from u by exploring edge (u, v).
- **Back edges** are those edges (u, v) connecting a vertex u to an *ancestor* v in a DFS tree.

  We consider self-loops, to be a back edges.

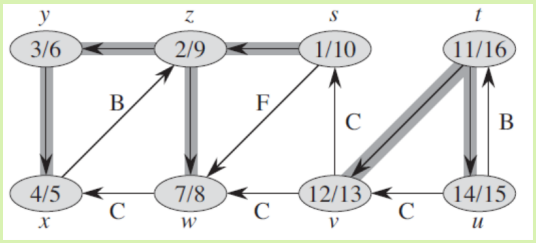


Examples of Tree edges (s, z), (z, y), (y, x), (t, v) and (t, u).

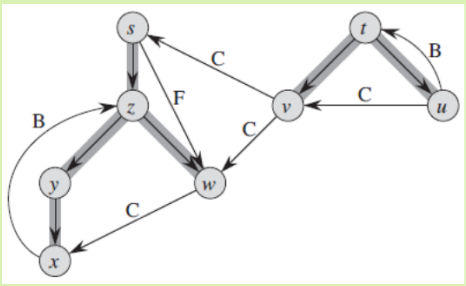
Examples of Back edges (x, z) and (u, t).

- The **key idea**: the color of a node when explored from an *edge*
  - The color is WHITE $\Rightarrow$ a tree edge.
  - The color is GRAY $\Rightarrow$ a back edge.

Classification of Edges II



- **Forward edges** are those *nontree* edges (u, v) connecting a vertex u to a descendant v in the DFS tree.
- **Cross edges** are all other edges. They can go between vertices in the same DFS tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different DFS tree.



Examples of a Forward edge (s, w).

Examples of Cross edges (w, x), (v, w), (v, s) and (u, v).

- The **key idea**: the color of a node when explored from an *edge*
  - The color is BLACK $\Rightarrow$ a forward or a cross edge.

Motivation for Topological Sorting or Ordering Algorithm I

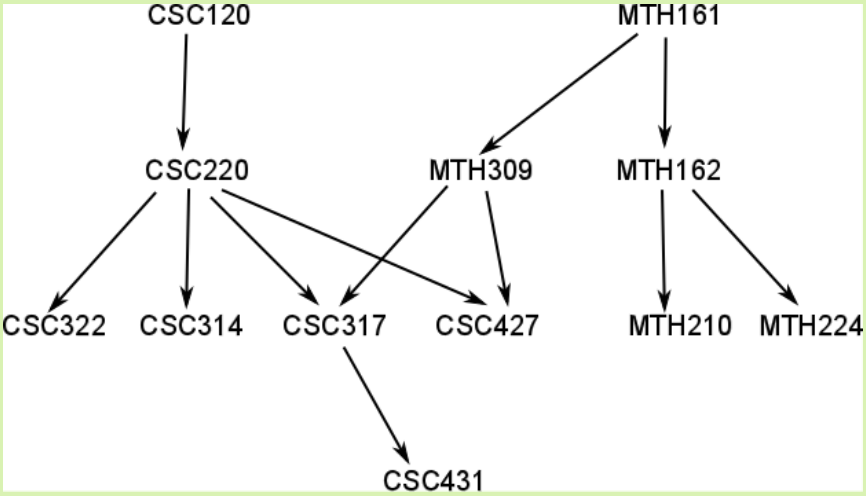
- Why we need topological sorting or ordering?
- The list of **required** course for a CS major;
  - Computer Science courses
    CSC120, CSC220, CSC314, CSC322, CSC317, CSC427, and CSC 431
  - Math courses
    MTH161, MTH162, MTH210, MTH224, and MTH309
  - CSC120 and MTH161 have no prerequisites (assuming a student has done enough math in high school)
  - A student wants to prepare a plan for completing both requirements

Table: Prerequisites for required CSC and MTH courses.

Course	Prerequisite(s)
CSC120	NONE
CSC220	CSC120
CSC322	CSC220
CSC314	CSC220
CSC317	CSC220, MTH309
CSC427	CSC220, MTH309
CSC431	CSC317
MTH161	NONE
MTH162	MTH161
MTH210	MTH162
MTH224	MTH162
MTH309	MTH161

Motivation for Topological Sorting or Ordering Algorithm II

- How to make a sequential list so that if a student take courses from left to right, then prerequisites are satisfied?
- Draw a prerequisite graph.



- Notice, this is a *directed acyclic graph* (DAG).
- If the graph had a cycle, no solution exists.

Topological Sorting or Ordering Algorithm

TOPOLOGICAL-SORT(G)

1 L: an empty linked-list

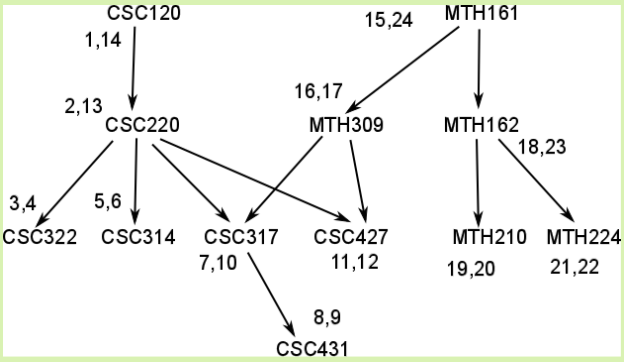
2 call DFS(G)

3 as each vertex is finished,

insert it onto the front of L

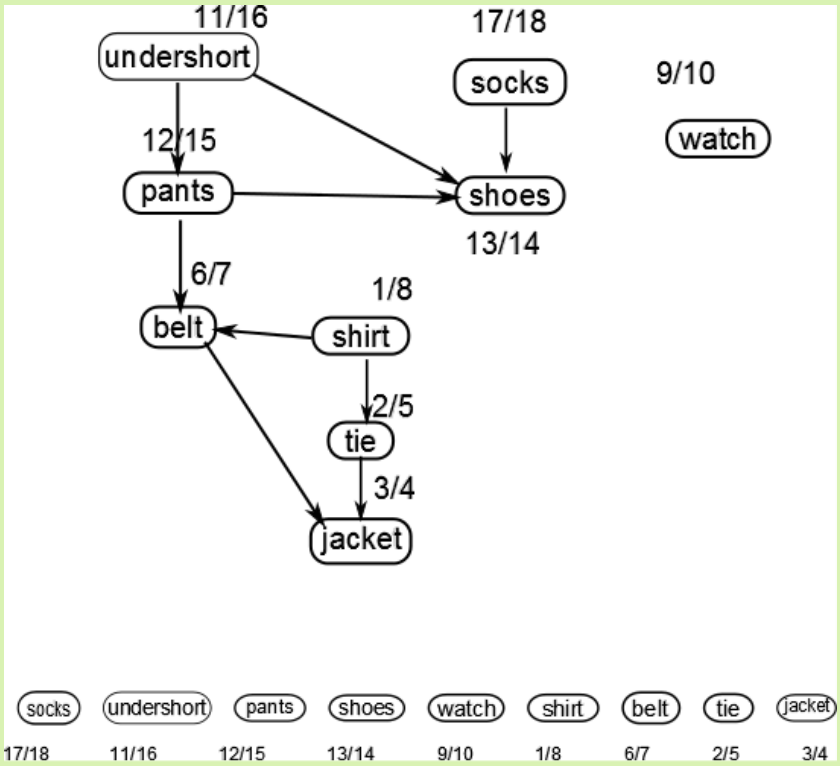
4 **return** the linked-list, L, of vertices

- Prerequisite Example:



- L = MTH161, MTH162, MTH224, MTH210, MTH309 CSC120, CSC220, CSC427, CSC317, CSC431,CSC314,CSC322

Topological Sorting or Ordering Algorithm: Example in the book



- Dressing dependence graph

  Each node has two numbers:

  first visit time

  final visit time
- The list below is sorted in decreasing order final visit time
- The order may not be unique for a DAG
- If you start with undershort, you will get a different list

Topological Sorting or Ordering Algorithm: Example in the book

Lemma (Lemma 22.11)

A directed graph G is acyclic if and only if a DFS og G yields no back edges.

Proof.

Outline: $\Rightarrow$ If there is a back edge (u, v), then v is an ancestor of u. Thus, there is a path from v to u and a directed edge from u to v, which completes a cycle.
$\Leftarrow$ If G contains a cycle, we have to show that there is a back edge. □

Theorem (Theorem 22.12)

TOPOLOGICAL-SORT *produces a topological sort of the directed acyclic graph provided as its input.*

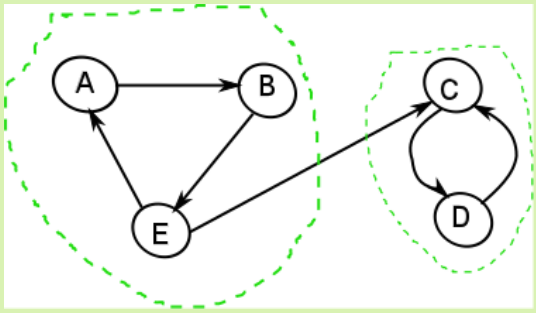
Proof.

Please read the book. □

Strongly connected components I

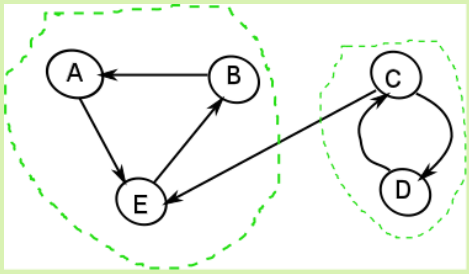
Transpose graph $G(V, E^T)$: For a directed graph $G(V, E)$ if direction of every edge $e = (u, v)$ is reversed, that is, $e^T = (v, u)$. And $E^T = \{(v, u) : \text{ such that } (u, v) \in E\}$. We will denote $G(V, E^T)$ by G^T

- A directed graph G



- In the graph G, there exits paths from vertices A, B, and E to vertices C or D. But no ptah to vertices A, B, and E from vertices C or D.

- Transpose graph G^T of the graph G



- Direction of every edge of G is reversed to obtain G^T.
- In G^T there is no path from vertices A, B, and E to vertices C or D.
- But in G^T there is path to vertices A, B, and E from vertices C or D.

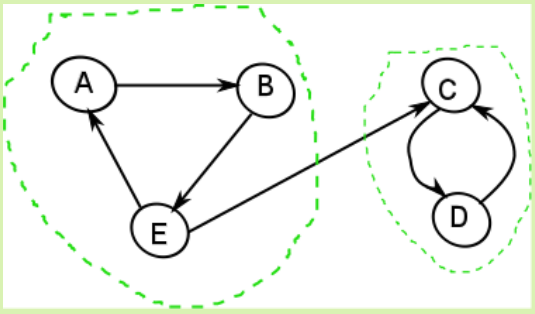
Strongly connected components II

Strongly connected graph (SCG): A directed graph is **strongly connected**, if there is a directed path from u to v, then there is also a directed path from vertex v to u for every vertex pair of u and v.

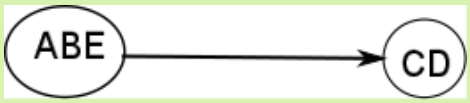
All directed graph may not be SCGs. There are many examples: social networks, software module dependency graph, food web in a ecosystem, and connections of neurons in our brain.

Strongly connected component (SSC) of a directed graph $G(V, E)$ is maximal set of vertices $C \subseteq V$, such that C forms a SCG.

A directed graph with two SCC



SCC graph



An Algorithm for finding strongly connected components

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex v

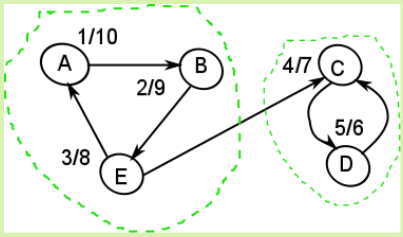
2 compute G^T

3 call DFS(G^T, but in the main loop of DFS, CONSIDER THE VERTICES in order of decreasing $u.f$ (as computed in line 1)

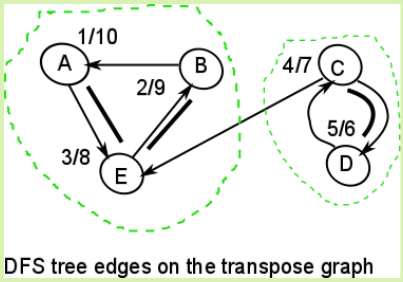
4 output the vertices of each tree in the depth-first forest formed in line 3 as aseparate strongly connected components

Trace of Execution of SCC algorithm

Graph G showing discovery times $u.d$ and finishing times $u.f$



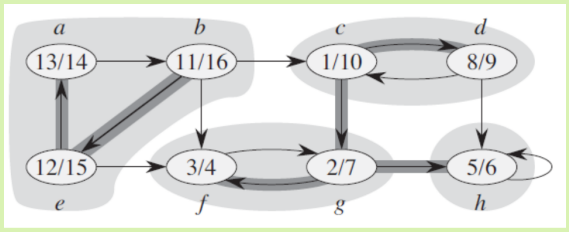
Transpose graph G^T with high-lighted DFS forest; vertices are searched with decreasing $u.f$.

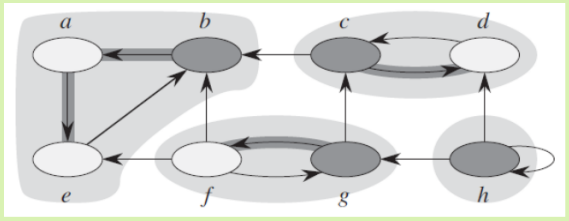


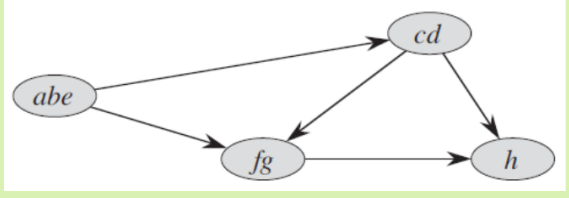
DFS tree edges on the transpose graph

Strongly connected components

Example in the textbook

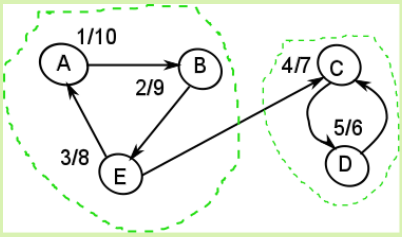




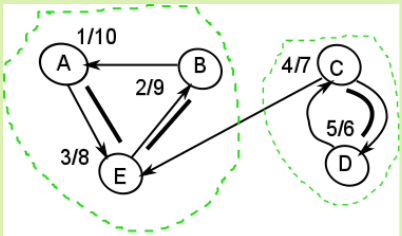


Trace of Execution of SCC algorithm

Graph G showing discovery times $u.d$ and finishing times $u.f$

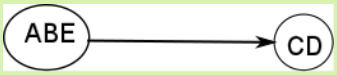


Transpose graph G^T with high-lighted DFS forest; vertices are searched with decreasing $u.f$.



DFS tree edges on the transpose graph

Strongly connected components



Example in the textbook