SEARCHING

What is Searching?

- Searching
  - retrieving information from a large amount of previously stored information

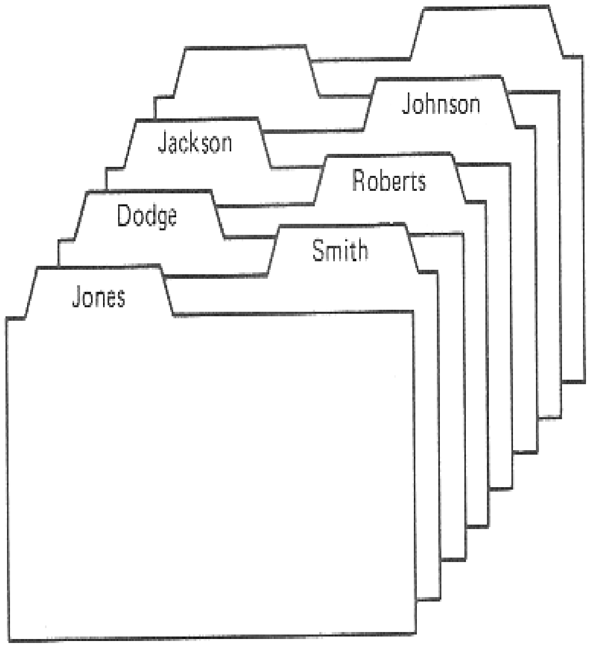
- What are the applications of searching?
  - Banking:
    - keep track of all customers' account balances and to search through them to check for various types

      of transactions

  - Transcript and Timetable:
  - Appropriate Route:
  - Street Directory:
  - Search engine: such as
    - need to look for relevant pages on the Web containing a given keyword

Searching (contd)

- Information are divided into records
- Each record has a key
- The goal of the search is to find all records with keys matching a given search key



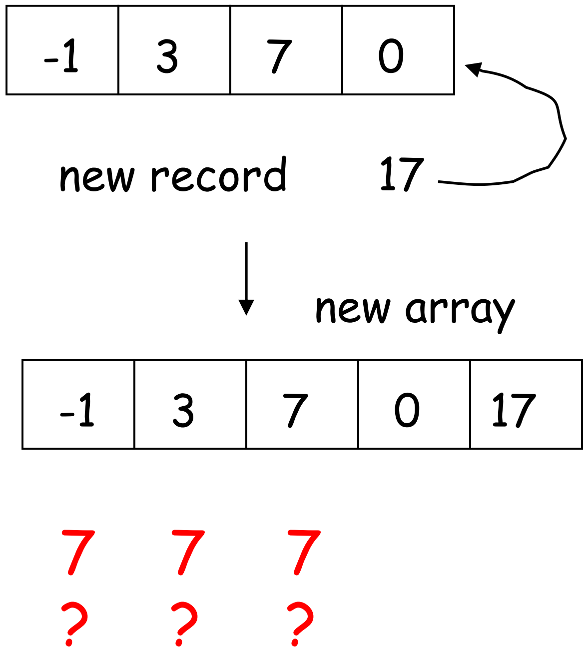
Records & their keys

Searching Methods

- **Elementary searching methods**
  - **Sequential (Linear) search**
  - **Binary search**

Sequential Searching

- The simplest method for searching is to store the records in an array
- When a new record is to be inserted
  - Put it at the end of the array
- When a search is performed
  - Look through the array sequentially



Sequential Search: Pseudocode

```
Sequential_search (L,key) {
    for (k = 1 to L.last) {
        if (key == L[k]) // found
            return k
    }
    return -1 // not found
}
```

Worst-case time complexity

worst case occurs when

key appears in the last position of array or

key is not in array

-1	3	7	0	17

17?

2?

Need to search all elements in array (n elements in array)

Hence complexity is O(n)

Binary Search

- Use to search for an item in a sorted array
- Input: an array L sorted in non-decreasing order, i.e.
  $L[1] \le L[2] \le \ldots \le L[n-1] \le L[n]$
- The binary-search algorithm begins by computing the midpoint $k = \left\lfloor \frac{1+n}{2} \right\rfloor$

example

L[1]	L[2]	L[3]	L[4]	L[5]

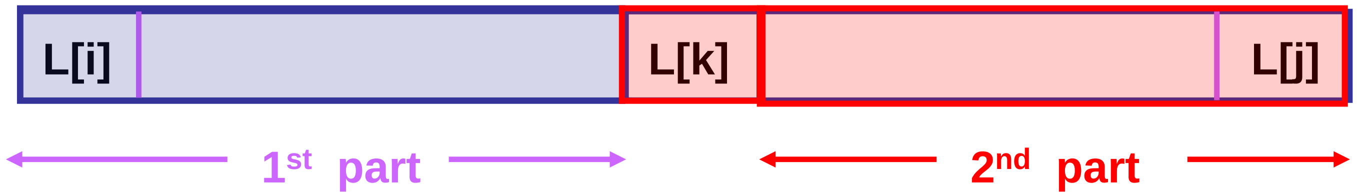
$$k = \left\lfloor \frac{1+5}{2} \right\rfloor = 3$$

- If L[k] = key, the problem is solved (record is found!)
- Otherwise array is divided into two parts of nearly equal size:

Array 1: L[1], L[2], …, L[k-1]   Array 2: L[k+1], L[k+2], …, L[n]

- If key < L[k], then it must be in array ?   => we ignore Array 1 or 2?
- If key > L[k], then search for the key in Array 1 or 2?

Binary Search




```
bsearch(L,i,j,key) {
    while (i ≤ j) {
        k = (i + j)/2    // midpoint
        if (key == L[k]) // found
            return k
        if (key < L[k]) // search first part
            j = k - 1
        else // search second part
            i = k + 1
    }
    return -1 // not found
}
```

Binary Search

```
bsearch(L,i,j,key) {
    while (i ≤ j) {
        k = (i + j)/2 // midpoint
        if (key == L[k]) // found
            return k
        // search first part
        if (key < L[k])
            j = k - 1
        else // search second part
            i = k + 1
    }
    return -1 // not found
}
```

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=51, i=1, j=8

Loop 1

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=51 , k=4 , i=5 , j=8

Loop 2

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=51 , k=6 , i=7 , j=8

Loop 3

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=51 , k=7 , i=8 , j=8

Loop 4

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=51 , k=8

Binary Search

```
bsearch(L,i,j,key) {
    while (i ≤ j) {
        k = (i + j)/2 // midpoint
        if (key == L[k]) // found
            return k
        // search first part
        if (key < L[k])
            j = k - 1
        else // search second part
            i = k + 1
    }
    return -1 // not found
}
```

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=29, i=1, j=8

Loop 1

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=29 , k=4 , i=5 , j=8

Loop 2

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=29 , k=6 , i=5 , j=5

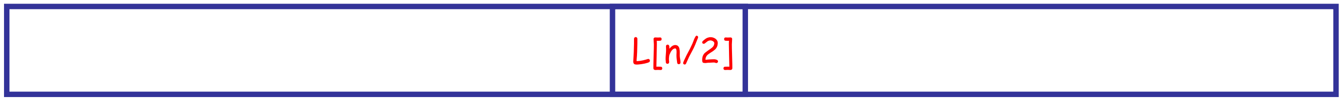
Loop 3

	1	2	3	4	5	6	7	8
L=	3	14	15	17	28	31	40	51

Key=29 , k=5 , i=6 , j=5

Binary Search: Worst-case Time Complexity

Sorted array L with n elements



Key < L[n/2]   Key > L[n/2]

- Let b = time required for key comparisons
- T(n) = worst-case time complexity

$$
\begin{aligned}
T(n) &= b + T(n/2) \\
&= b + b + T(n/4) \\
&= b + b + b + T(n/8) \\
&= kb + T(n/2^k) \\
&= \dots \\
&= b*\log(n) + T(1) \\
&= b*\log(n) + b \\
&= b(\log(n)+1). \text{ Thus, } O(\log(n))
\end{aligned}
$$

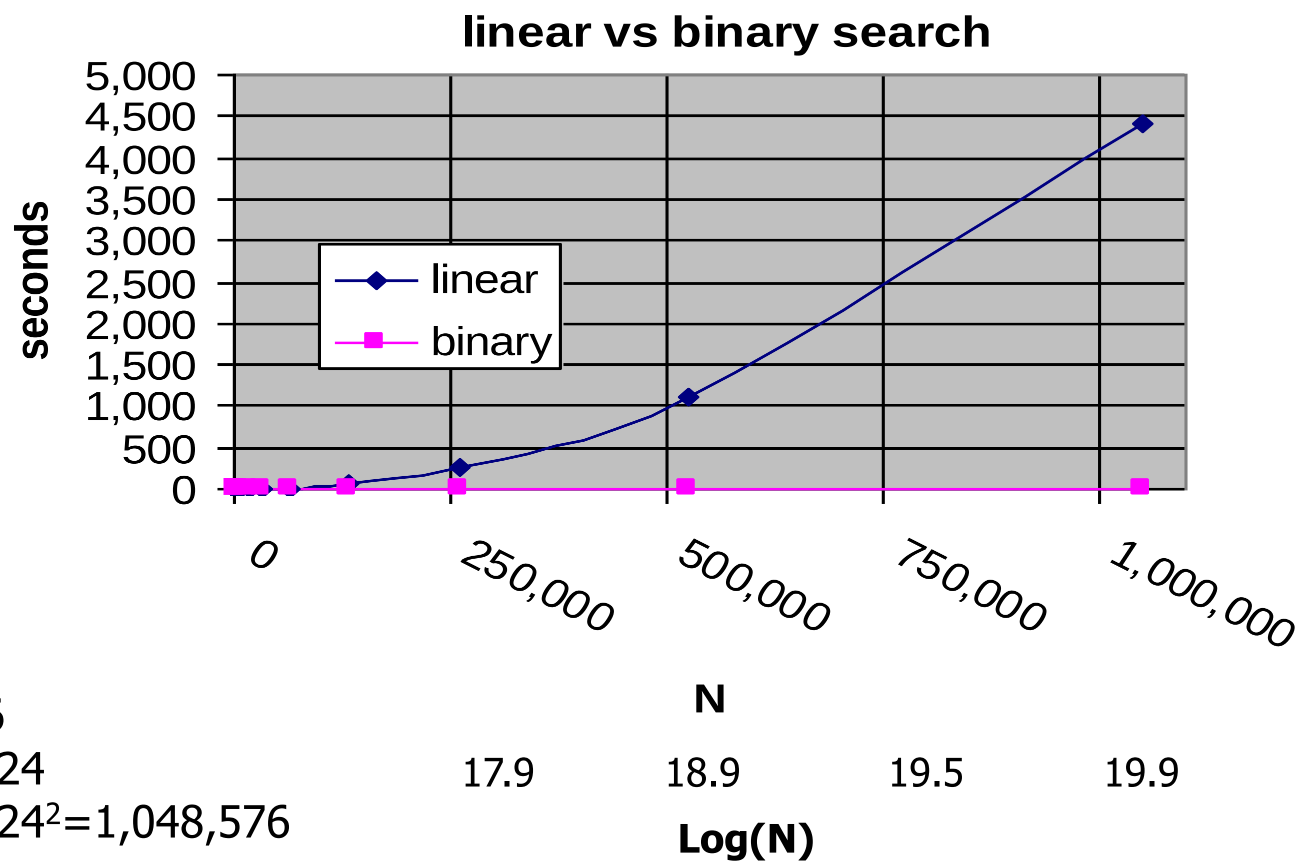
$n/2^k = 1 \Rightarrow 2^k = n$

$\Rightarrow k = \log(n)$

Sequential Search vs Binary Search

Sequential (linear) search : O(n)

Binary search : O(log(n))



$2^1=2$

$2^2=4$

$2^4=16$

$2^8=256$

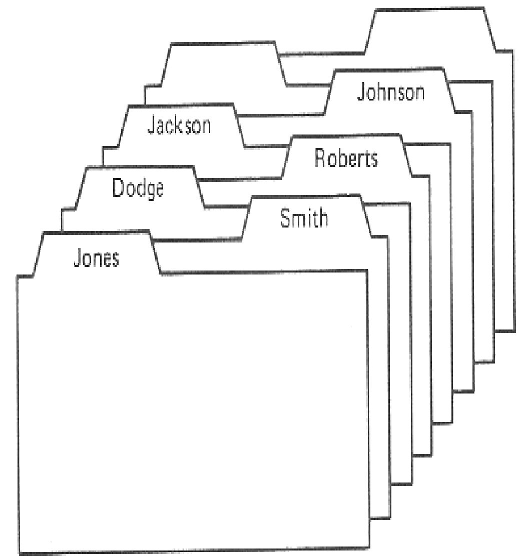
$2^{10}=1024$

$2^{20}=1024^2=1,048,576$

Suppose time required for 1000 key comparisons is capped by 4.5 seconds

Searching in Multi-dimensional Info

-1	3	7	0	17



Employees : Table

Employee ID	Last Name	First Name	Title	Title Of	Birth Date	Hire Date	Address	City
1	Davolio	Nancy	Sales Representative	Ms.	08-Dec-1968	01-May-1992	507 - 20th Ave. E.	Seattle
2	Fuller	Andrew	Vice President, Sales	Dr.	19-Feb-1952	14-Aug-1992	908 W. Capital Way	Tacoma
3	Leverling	Janet	Sales Representative	Ms.	30-Aug-1963	01-Apr-1992	722 Moss Bay Blvd.	Kirkland
4	Peacock	Margaret	Sales Representative	Mrs.	19-Sep-1958	03-May-1993	4110 Old Redmond Rd.	Redmond
5	Buchanan	Steven	Sales Manager	Mr.	04-Mar-1955	17-Oct-1993	14 Garrett Hill	London
6	Suyama	Michael	Sales Representative	Mr.	02-Jul-1963	17-Oct-1993	Coventry House	London
7	King	Robert	Sales Representative	Mr.	29-May-1960	02-Jan-1994	Edgeham Hollow	London
8	Callahan	Laura	Inside Sales Coordinator	Ms.	09-Jan-1958	05-Mar-1994	4726 - 11th Ave. N.E.	Seattle
9	Dodsworth	Anne	Sales Representative	Ms.	02-Jul-1969	15-Nov-1994	7 Houndstooth Rd.	London

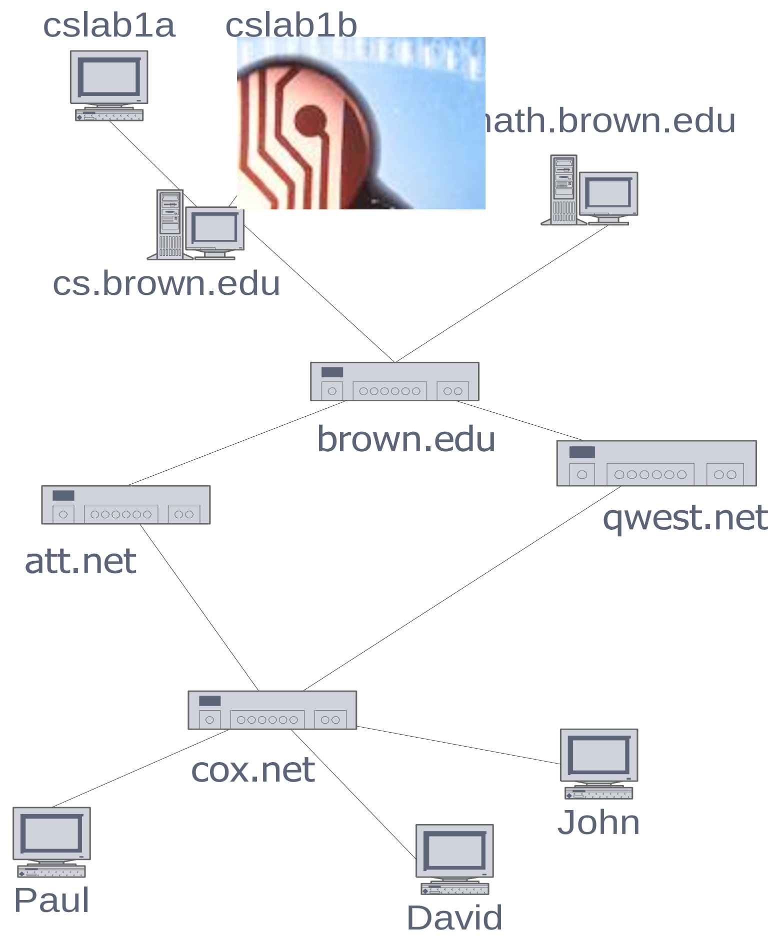
Learning Takeaway

- **Binary search uses Divide and Conquer**
  - **splits array into two halves**
  - **continue search only on relevant half**
  - **much more efficient than linear search**

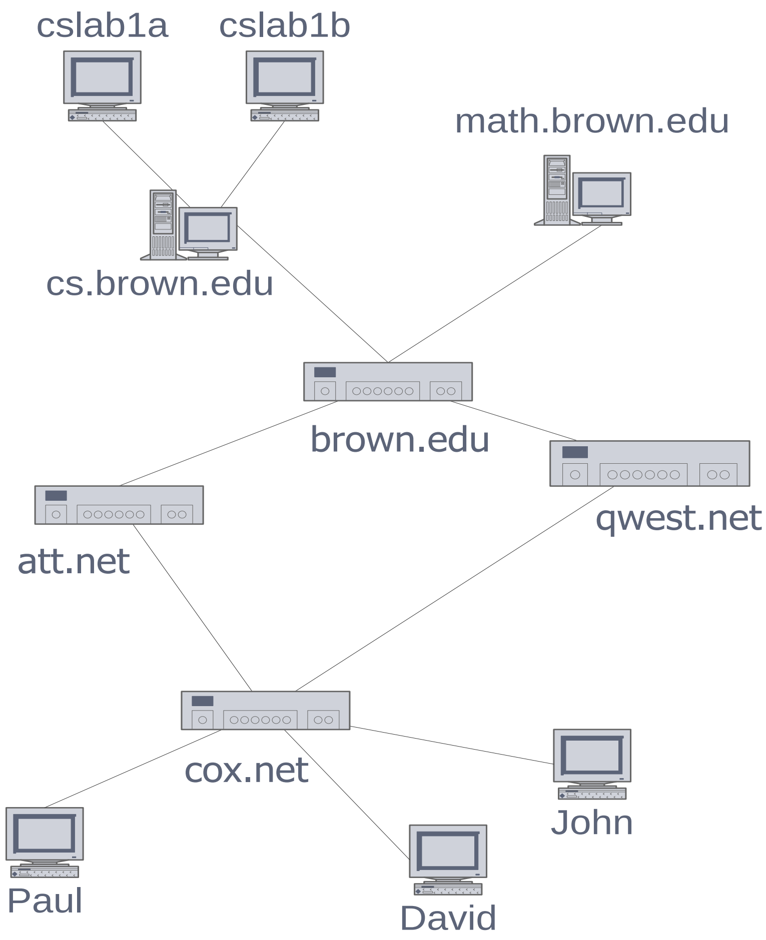
Graph Algorithms

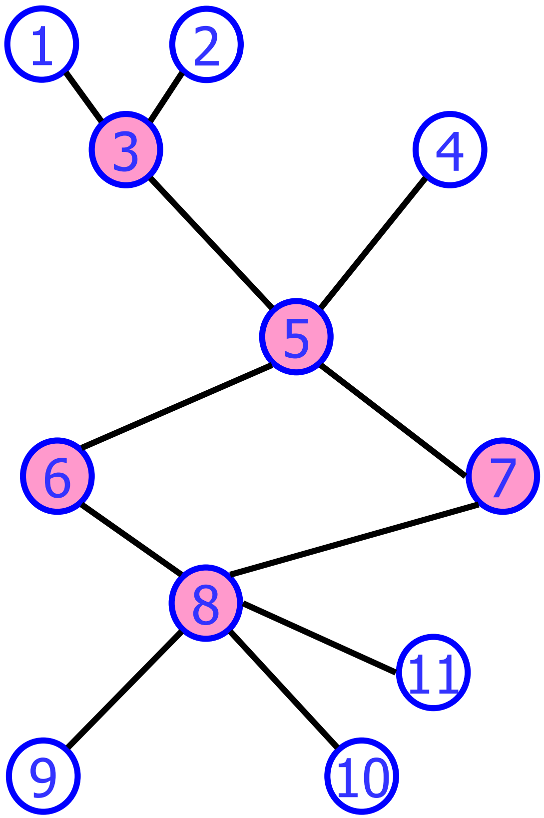
Applications

- **Electronic circuits**
  - ☞ **Printed circuit board**
  - ☞ **Integrated circuit**
- **Transportation networks**
  - ☞ **Highway network**
  - ☞ **Flight network**
- **Computer networks**
  - ☞ **Local area network**
  - ☞ **Internet**
  - ☞ **Web**



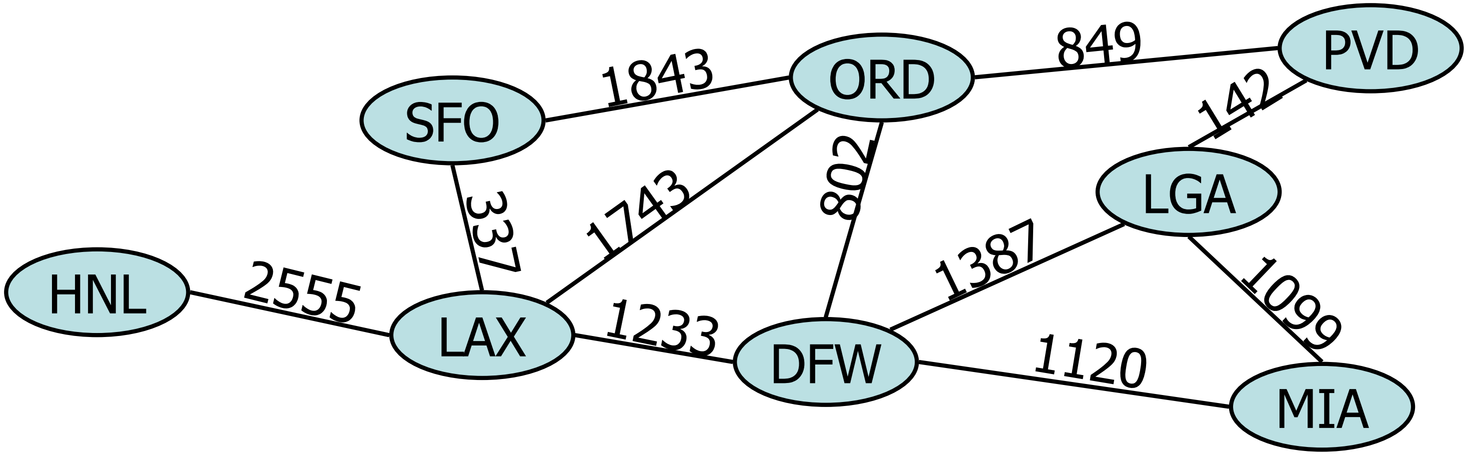
Applications



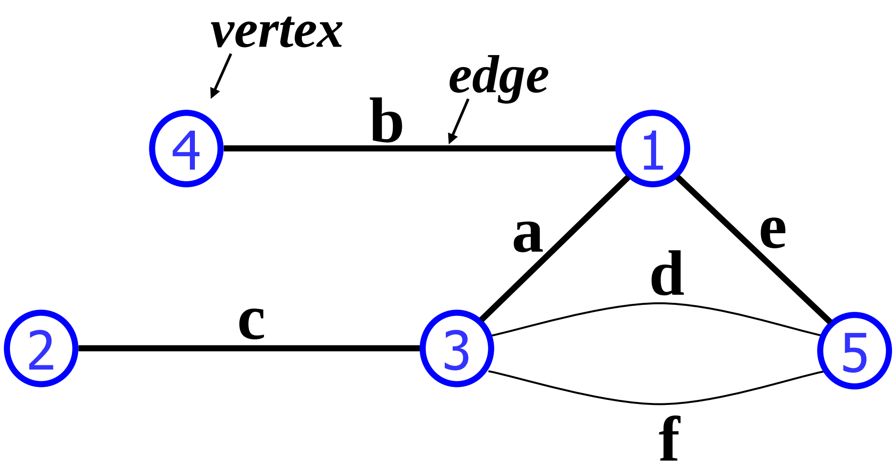


Graph

- A graph **G** is a pair (V, E), where
  - V is a set of nodes, called **vertices**
  - E is a collection of pairs of vertices, called **edges**
  - We write G = (V,E)
- **Example:**
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



- **An undirected graph contains only bi-directional links**
  - **each edge is associated with an unordered pair of vertices**
    - **if e is an edge connecting vertices u & v, then we write e = (u,v) or e = (v,u)**



G = (V,E)

V = {1,2,3,4,5}

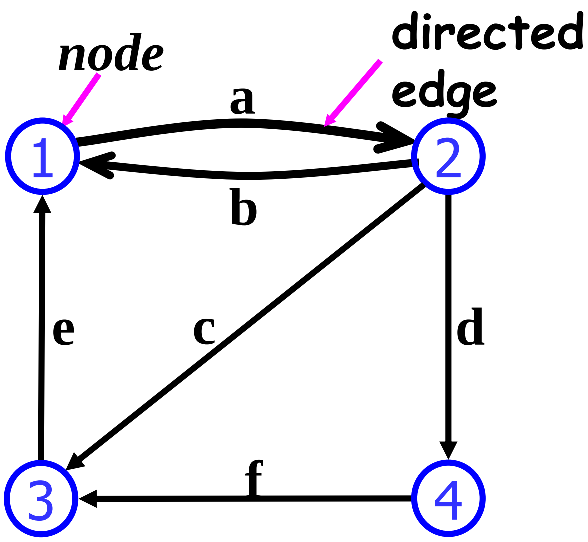
E = {a,b,c,d,e,f}

a = (1,3), b=(1,4), c=(2,3), etc

Application: Can be used to model a street system where each street is a two-way street

Directed Graph

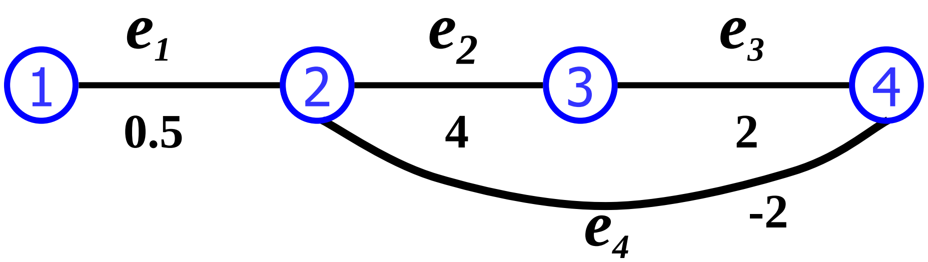
- **A directed graph is a graph containing uni-directional edges**
  - **each edge is associated with an ordered pair of vertices**
    - **if e is a directed edge connecting vertices u & v, then we write e = (u,v)**



Can be used to model a street system where each street is a one-way street

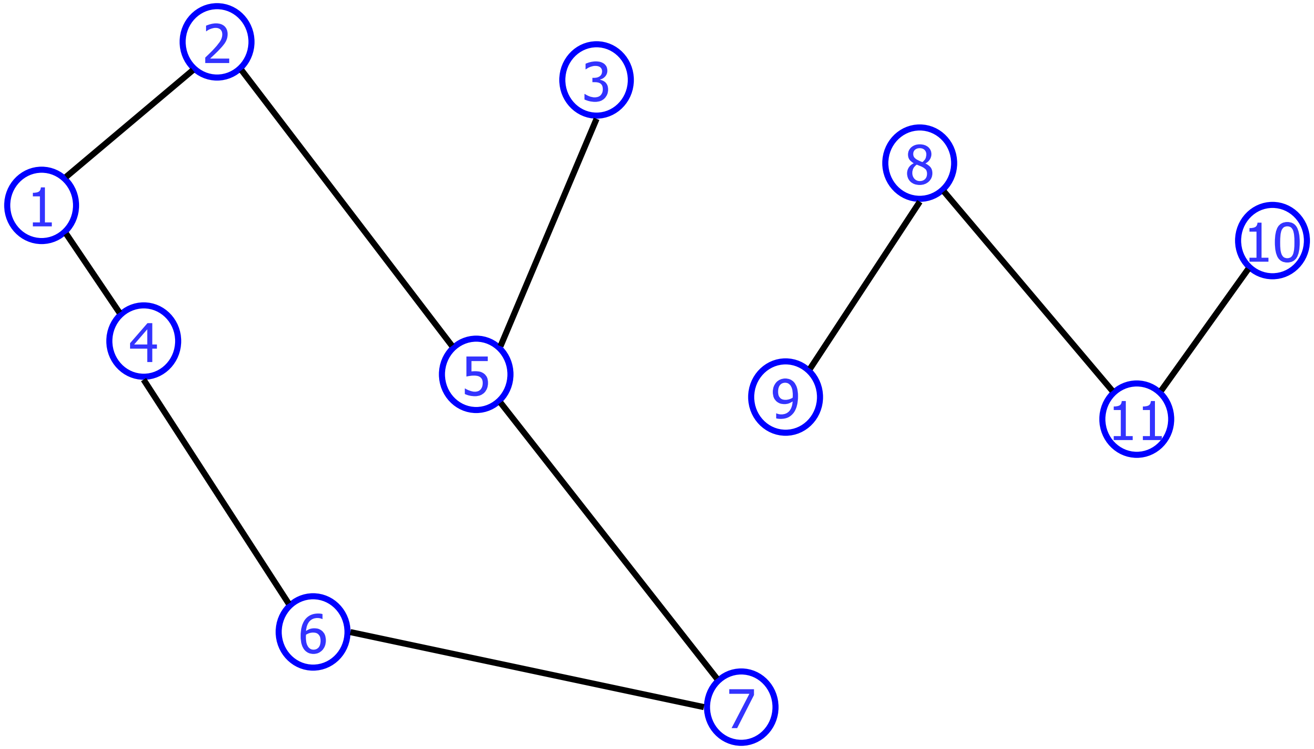
Weighted Graphs

- A **weighted graph** is a graph where each edge is associated with a number (value). An example is as follows



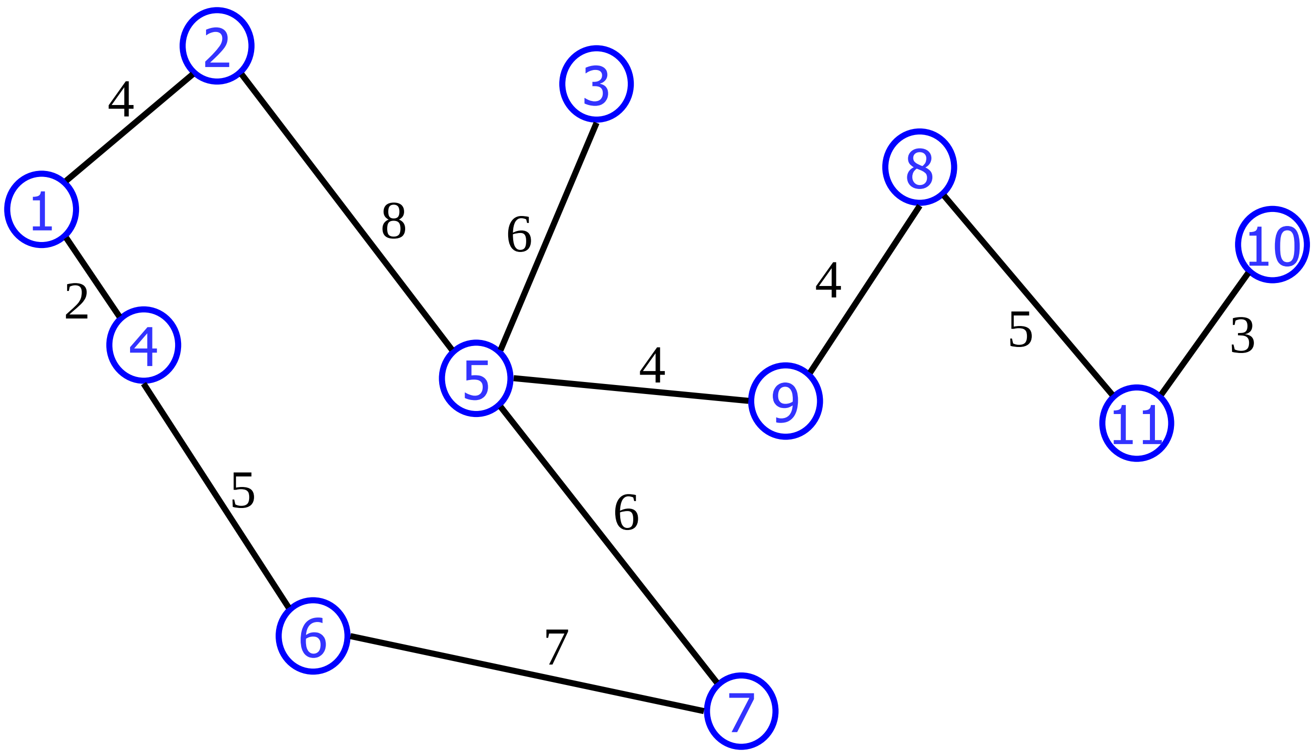
- The actual meanings of the numbers depend on the application. In general they may be positive or negative.

Applications—Communication Network



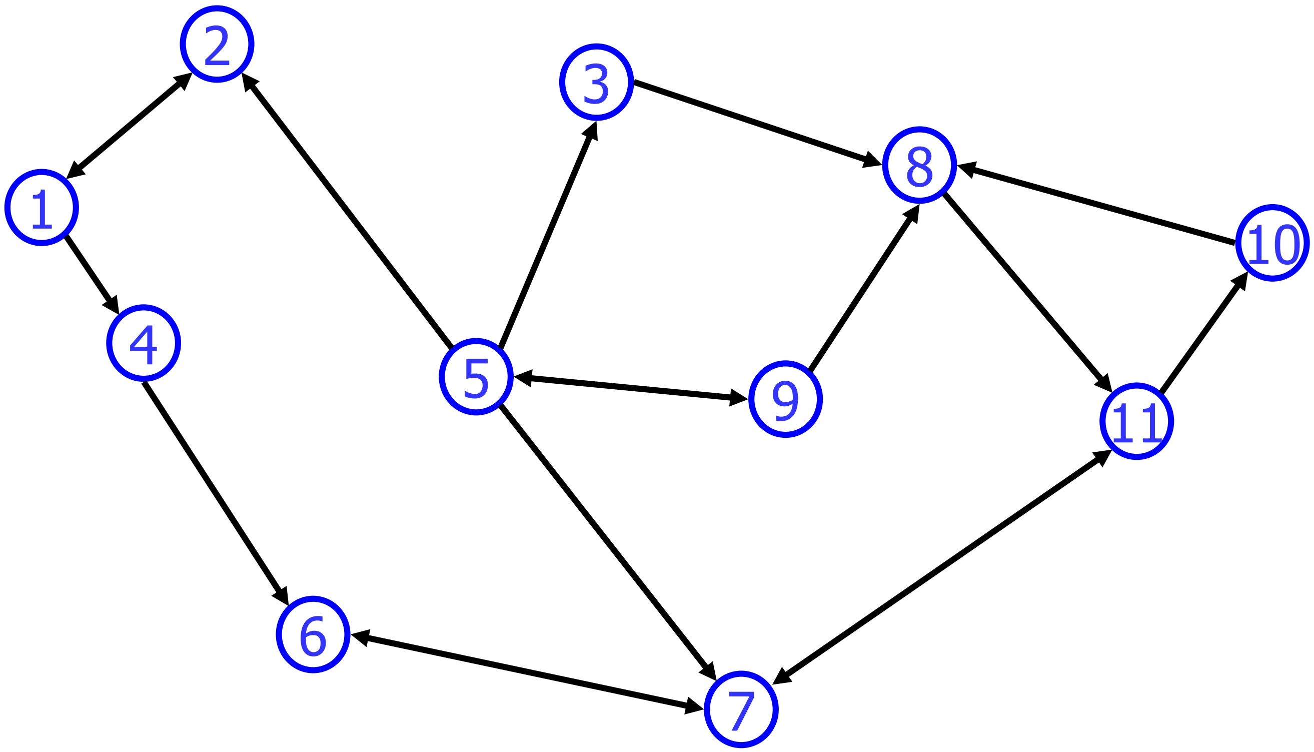
- Vertex = city, edge = communication link.

Driving Distance/Time Map



- Vertex = city, edge weight = driving distance/speed.

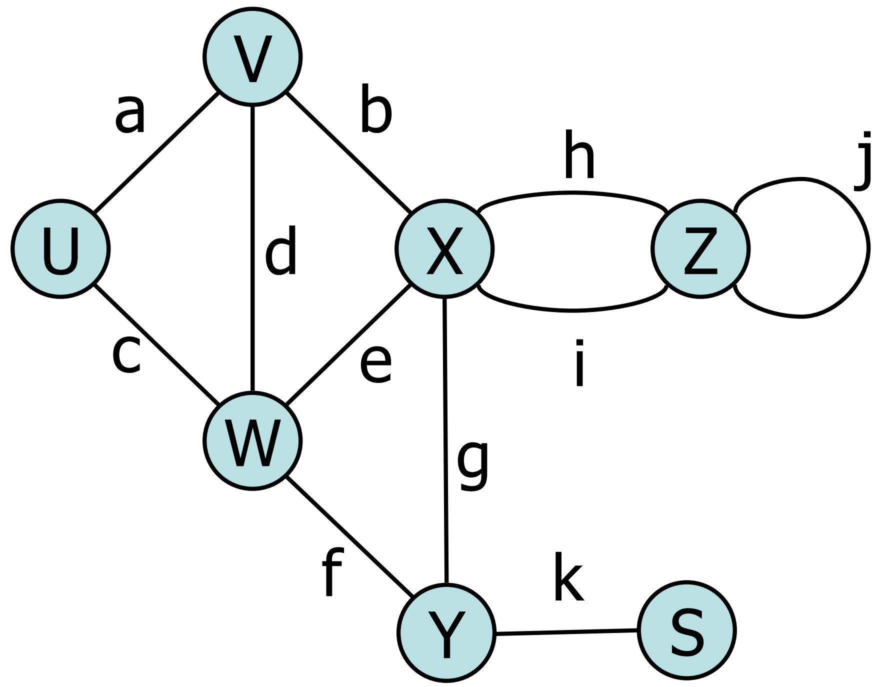
Street Map



- Some streets are one way.

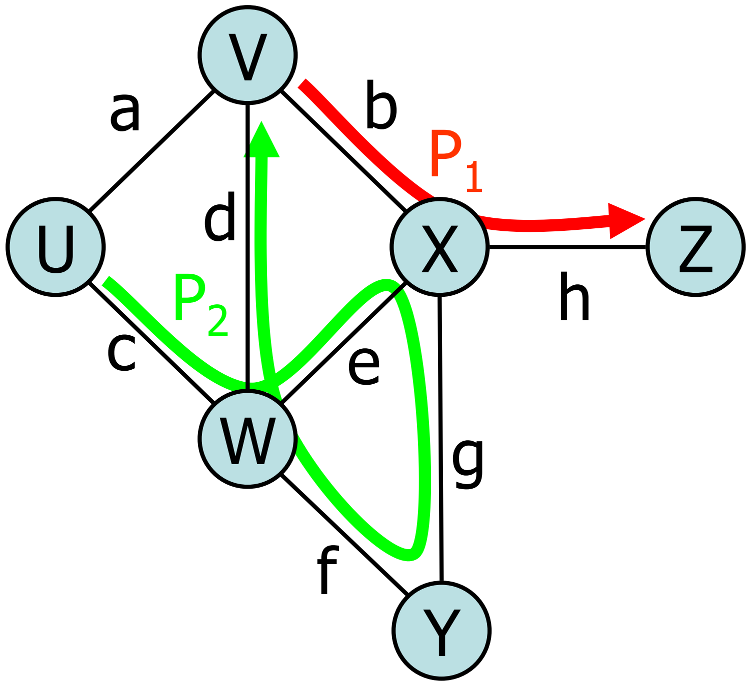
Terminology

- **End vertices (or endpoints) of an edge**
  - ☞ **U and V are the endpoints of a**
- **Edges incident on a vertex**
  - ☞ **a, d, and b are incident on V**
- **Adjacent vertices**
  - ☞ **U and V are adjacent**
- **Degree of a vertex**
  - ☞ **X has degree 5**
- **Leaf – vertex with degree 1**
  - ☞ **S is a leaf**
- **Parallel edges**
  - ☞ **h and i are parallel edges**
- **Self-loop**
  - ☞ **j is a self-loop**



Terminology (cont.)

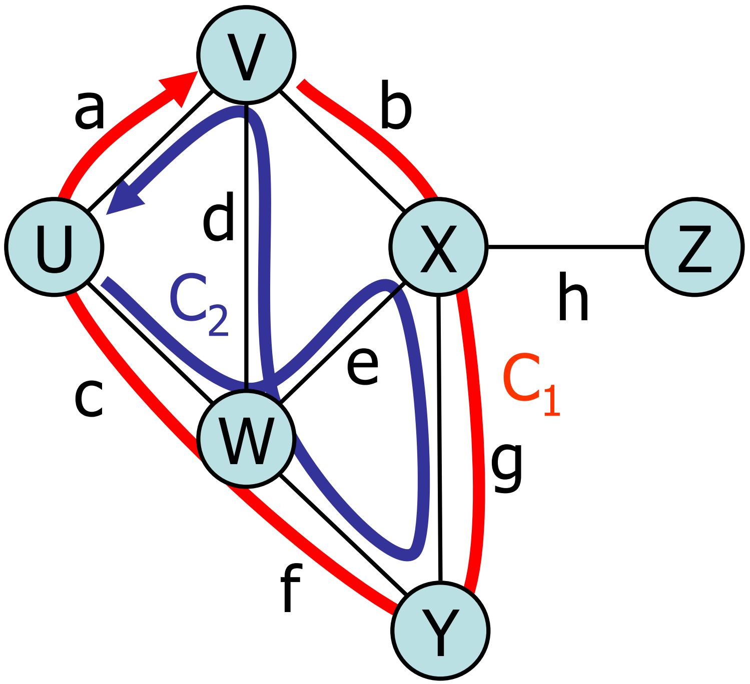
- **Simple graph**
  - A graph with neither loops nor parallel edges
- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- **Simple path**
  - path with no repeated vertices
- **Examples**
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



If graph is simple, can represent a path by just listing the vertices.

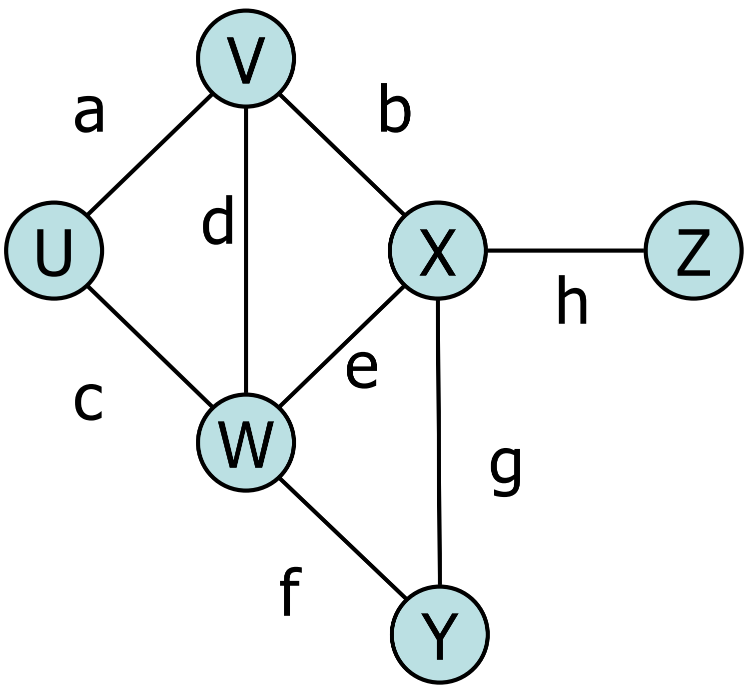
Terminology (cont.)

- **Cycle**
  - **A cycle is a path whose initial vertex and terminal vertex are identical and there are no repeated edges**
- **Simple cycle**
  - **cycle with no repeated vertices**
- **Examples**
  - **C_1=(V,b,X,g,Y,f,W,c,U,a,V) is a simple cycle**
  - **C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,U) is a cycle that is not simple**

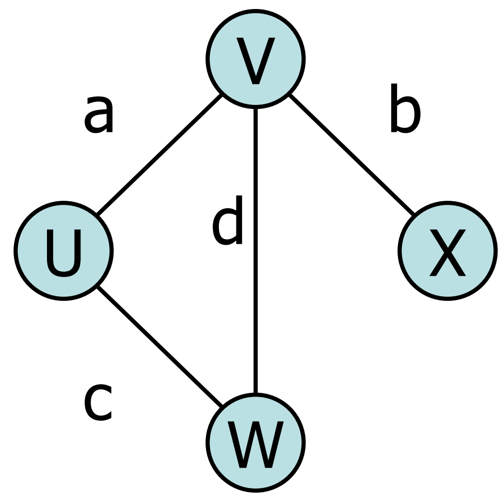


Subgraphs

- A graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if all its vertices and edges are in G, i.e., $V' \subset V$ and $E' \subset E$.



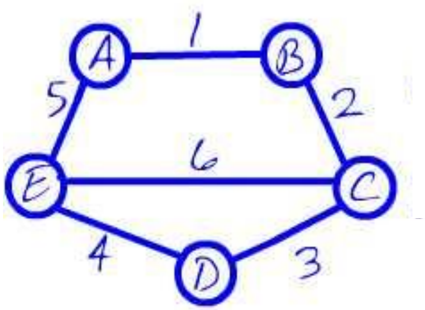
Graph G



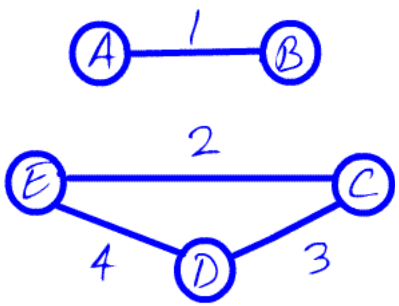
Subgraph of G

Connectivity

- **A graph is connected if there is a path joining every pair of distinct vertices; otherwise it is called disconnected.**



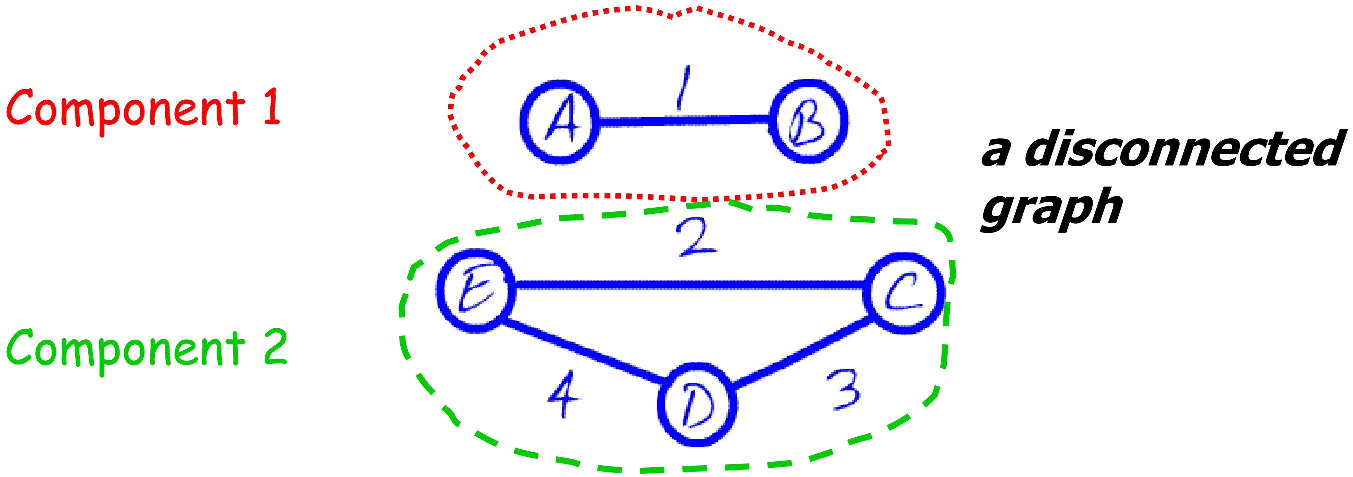
a connected graph



a disconnected graph

Components of a Graph

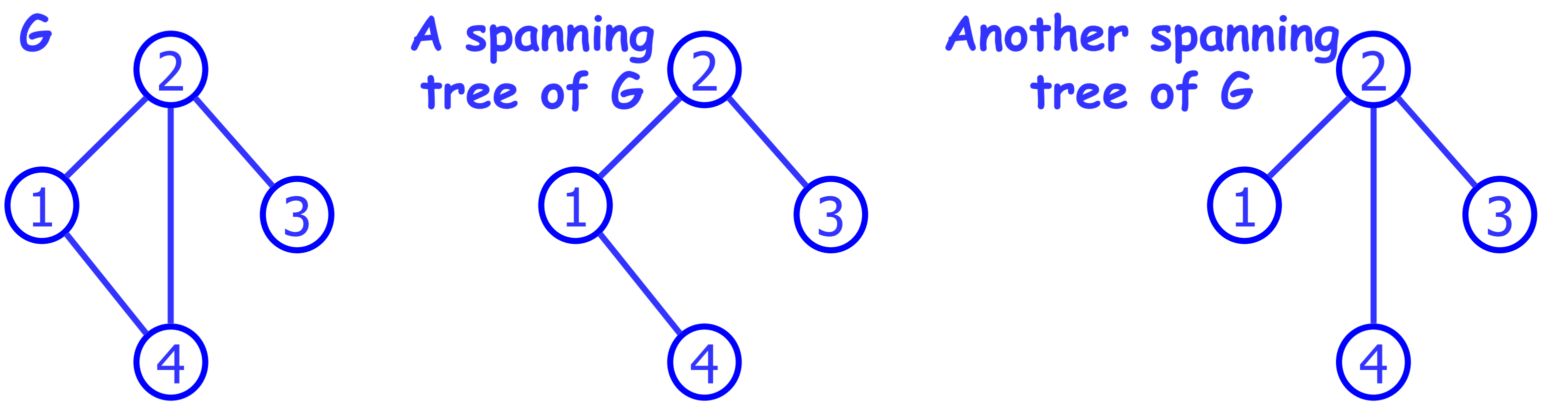
- The sets of nodes in a graph with paths to one another are (connected) components. The edges between these nodes are also part of the components.



A graph of two components

Trees

- A graph is called a **tree** if it is *connected* and it contains no cycle.
  - There is a unique path between two vertices in a tree
  - Any tree with n nodes will contain n-1 edges
- A **spanning tree** of a graph G is a subgraph of G that is a tree and that includes all vertices of G.
  - Every connected graph possesses (at least) one spanning tree



Trees

- **Every tree with at least 2 vertices has at least 2 leaves.**

Proof: A connected graph with at least 2 vertices has an edge. Consider a maximal path with at least 1 edge. The endpoints of this path have degree 1, otherwise we can add a new vertex to the path and make it longer, a contradiction.

Trees

- **Why will any tree with n nodes contain n-1 edges?**

If n=1, the tree contains zero edge

If n=2, the tree contains one edge

- **Proof:**

Basis Step: n =1, zero edge so claim is true.

Trees

- **Why will any tree with n nodes contain n-1 edges?**

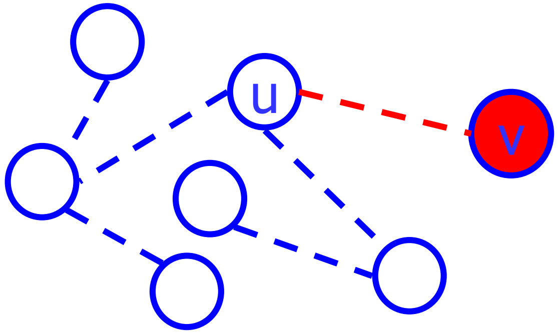
- **Proof:**

Inductive Step:

Assume n=k, the tree contains k-1 edges

For n=k+1

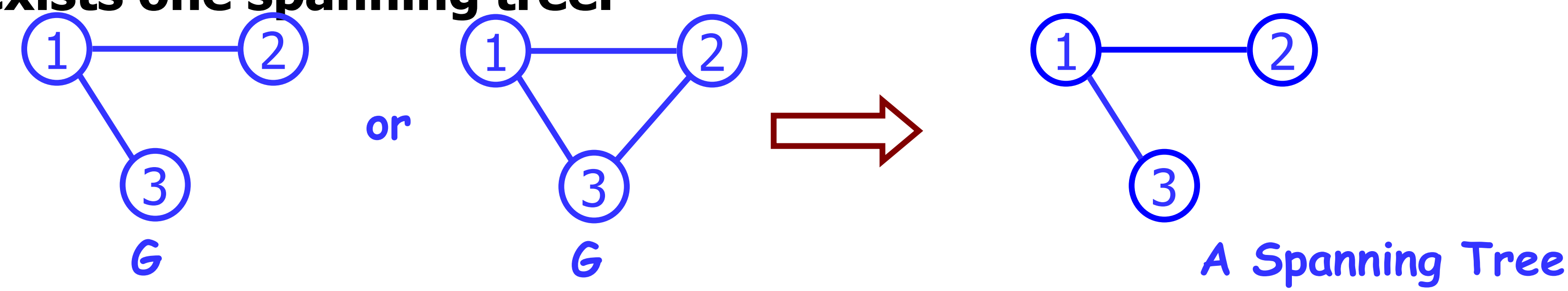
There is at least 1 leaf in the graph. Remove the leaf, and we get a tree T′ with k vertices. By induction hypothesis, T′ has k-1 edges, which implies that T has k-1+1 = k edges.



- **Why does every connected graph possess (at least) one spanning tree?**

For Graph *G* with n=2 vertices, there exists one spanning tree.

For Graph G with n=3 vertices, there also exists one spanning tree.



- **Proof:**

Basis Step: n=1, trivial.

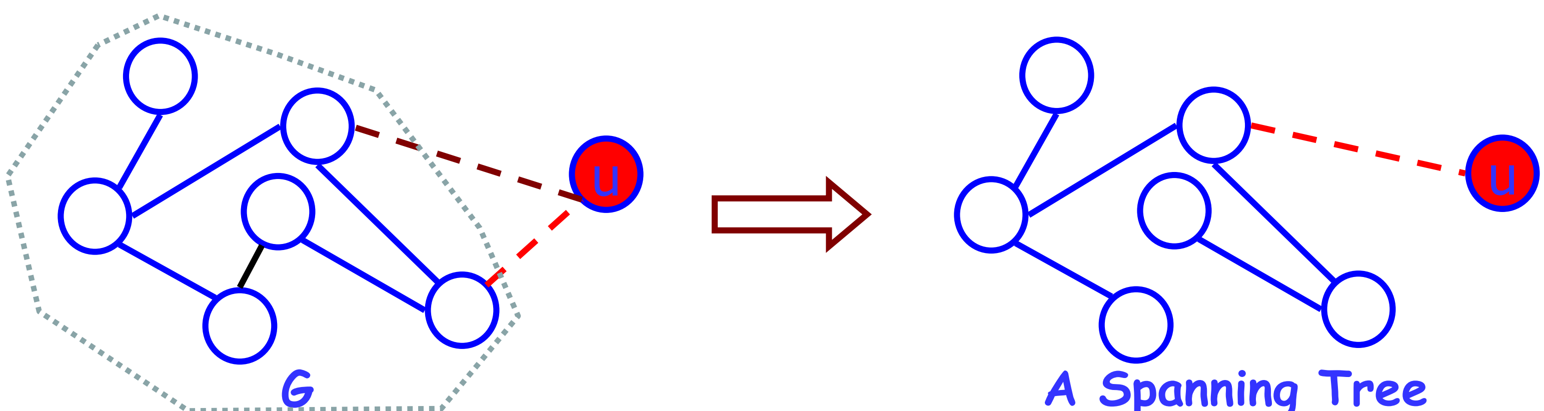
- **Why does every connected graph possess (at least) one spanning tree?**

- **Proof:**

Inductive Step:

Assume that for Graph G with n<=k vertices, there exists one spanning tree.

For Graph G with n=k+1 vertices. General idea:

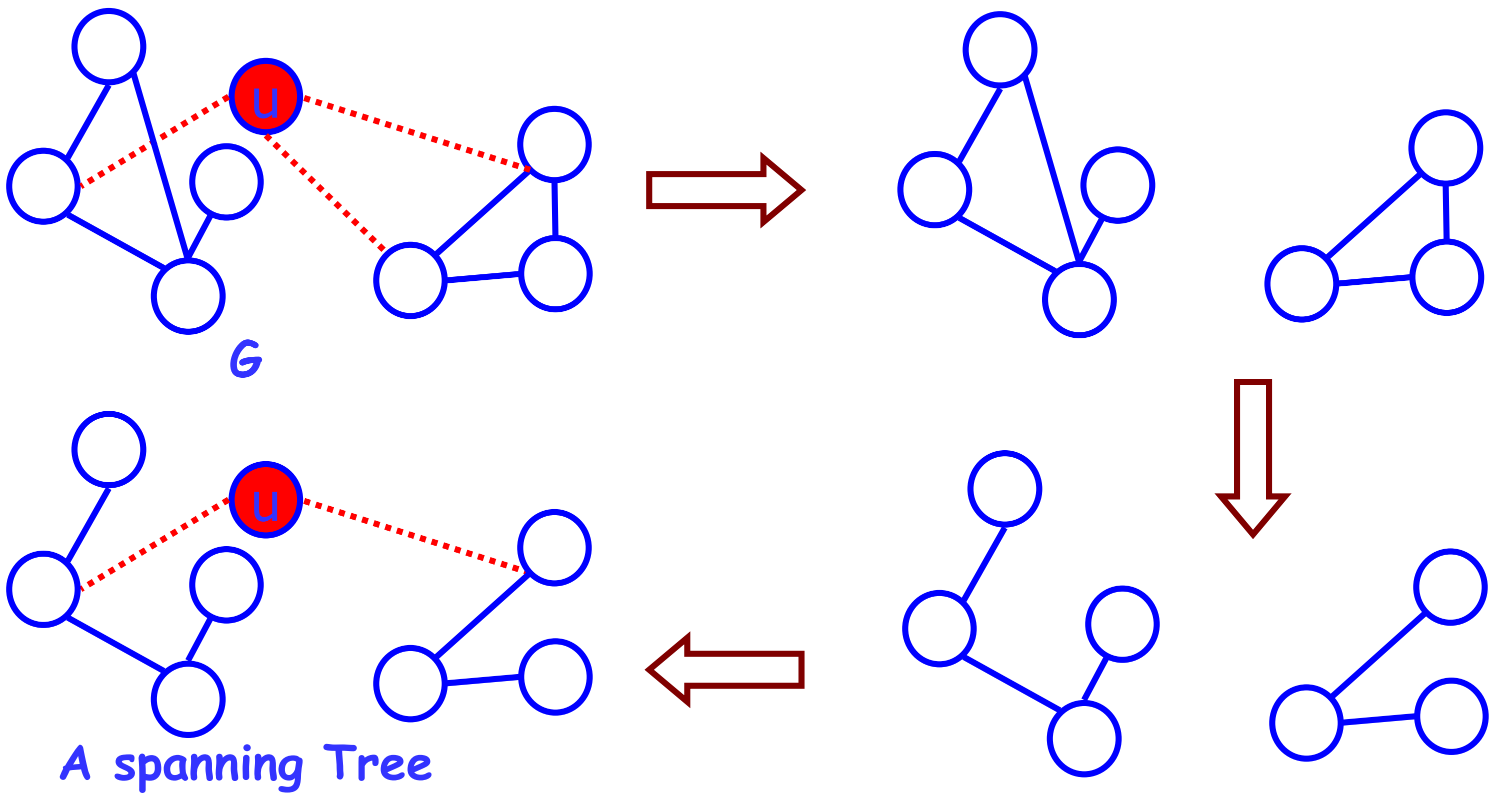


- **Proof:**

  Inductive Step:

  Assume that for Graph G with n<=k vertices, there exists one spanning tree.

  For for Graph G with n=k+1 vertices



Learning Takeaway

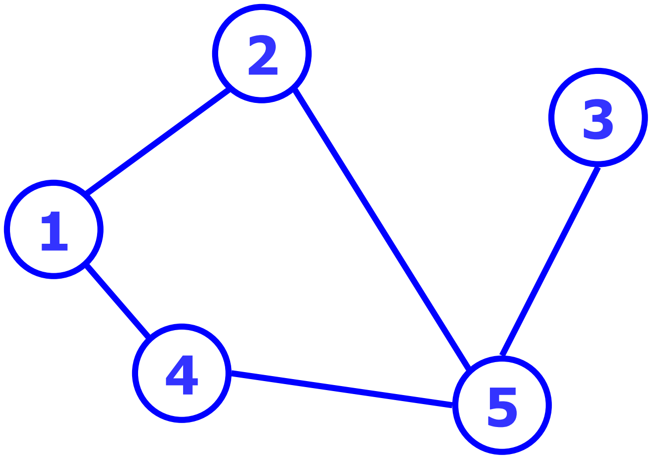
- **Basic graph theory terminologies.**

- **Mathematical induction is an often useful technique for proving graph related results.**
  - **induction on the number of vertices**
  - **induction on the number of edges**

- **A stronger form of mathematical induction is often required: e.g., in inductive step, need to assume claim is true for all $n \leq k$, and prove for $n = k + 1$.**

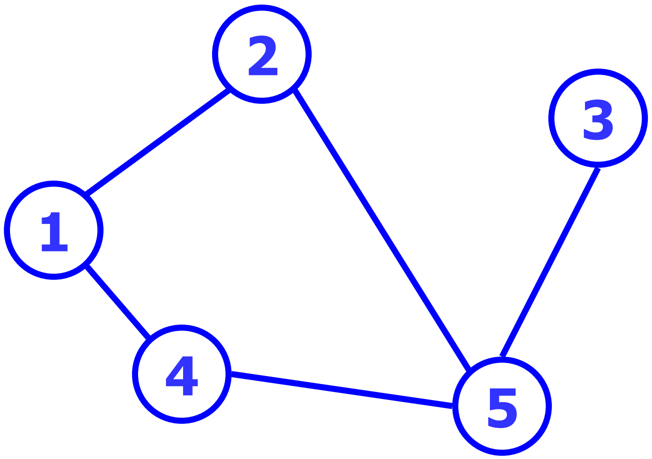
Graph Representations

Adjacency Matrix



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

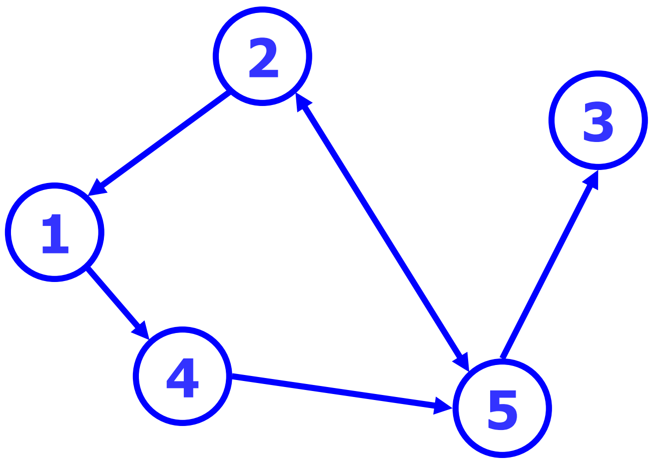
Adjacency Matrix Properties



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

- **Diagonal entries are zero.**
- **Adjacency matrix of an undirected graph is symmetric.**
  - ☞ **A(i,j) = A(j,i) for all i and j.**

Adjacency Matrix for Digraph



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

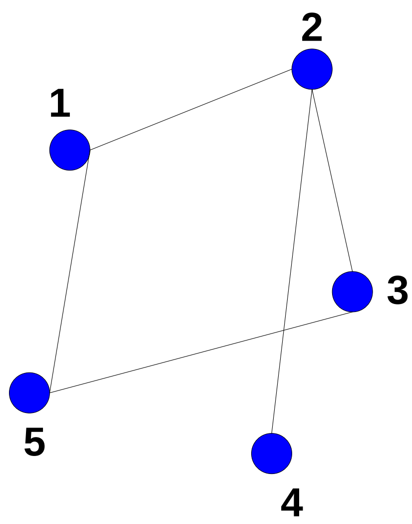
- ***A(i,j) = 1 iff (i,j) is an edge.***
- ***E.g. (1,4) is an edge but (4,1) is not.***

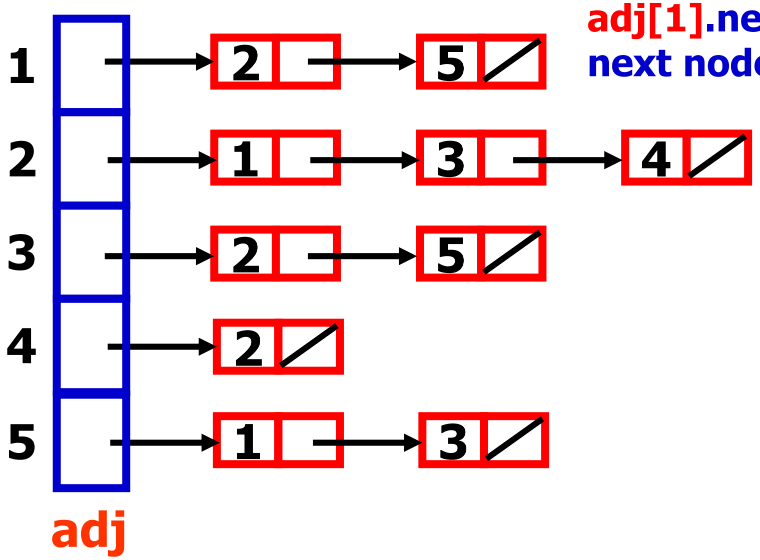
Adjacency Matrix

- n^2 bits of space
- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
  - $(n-1)n/2$ bits
- $O(n)$ time to find vertex degree and/or vertices adjacent to a given vertex.

Adjacency Lists

- Another way of representing a graph is to use linked lists
  - This is referred to as adjacency lists
- An array is used to access the various linked lists

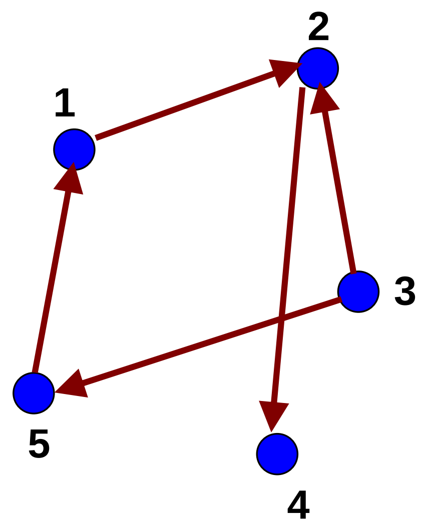


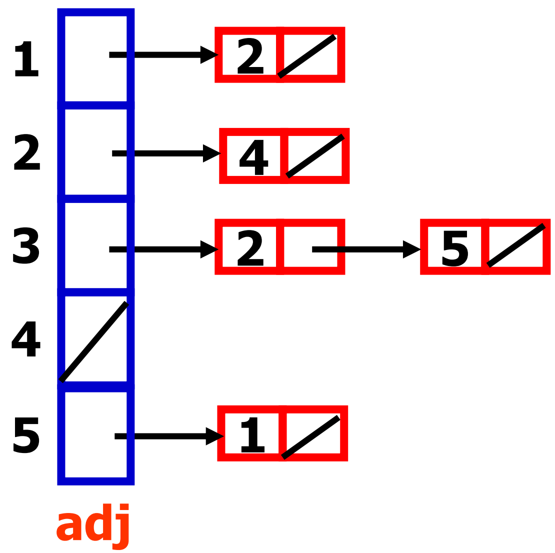


adj[1] is a pointer.
adj[1].data is 2
adj[1].next is pointer to next node in linked list.

Adjacency Lists for Digraphs

- Store neighbor to which the vertex has an outgoing edge to that neighbor.





adj[1] is a pointer.
adj[1].data is 2
adj[1].next is pointer to next node in linked list.

Let v be a vertex in a digraph.

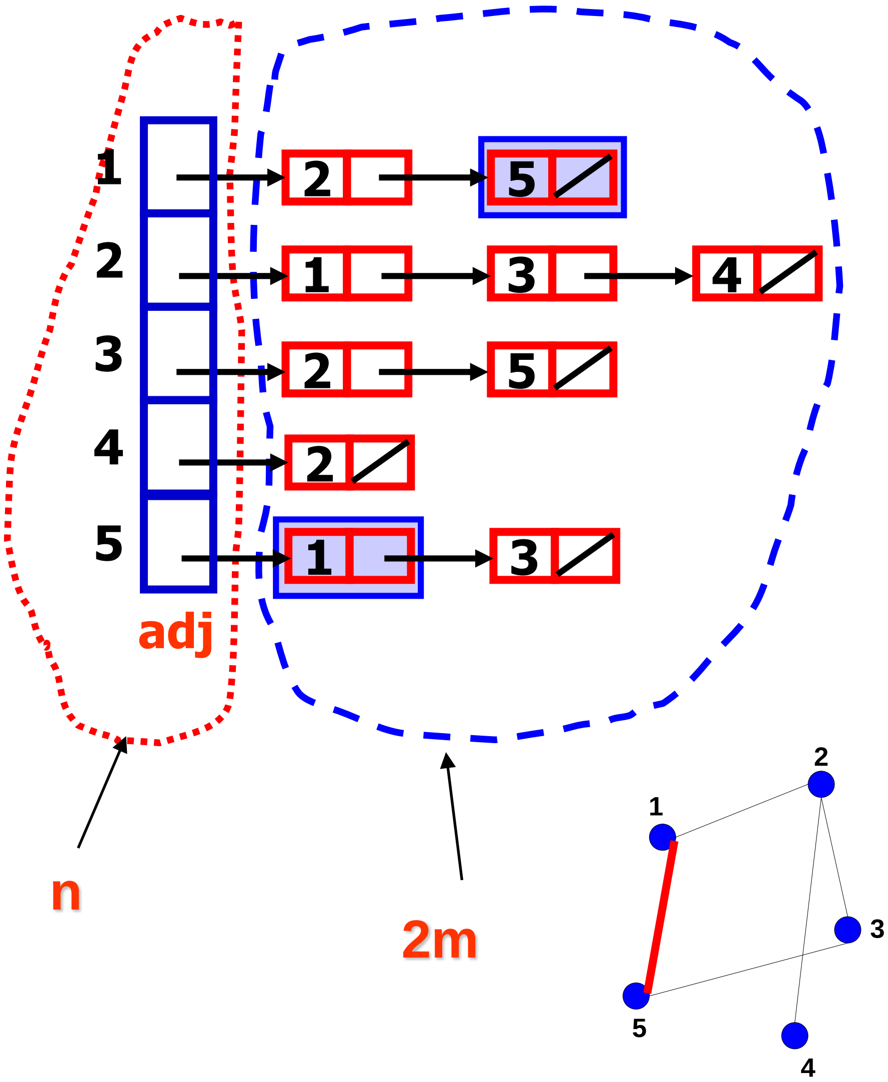
in-degree of v = number of incoming edges

out-degree of v = number of outgoing edges

in-degree of vertex 2 is 2

out-degree of vertex 2 is 1

Complexity of Adjacency Lists



- **Let m = number of edges in the graph**
- **Number of vertices = n**
- **Each edge (i,j) is represented twice in the adjacency lists: j appears once in vertex i's list and i appears once in vertex j's list**
- **Hence there is a total of 2m nodes in the adjacency lists**

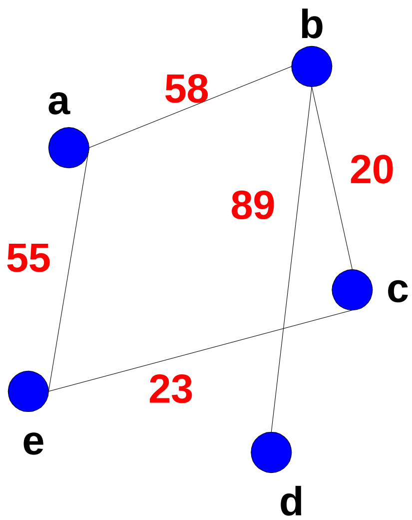
Space complexity = O(n+m)

Adjacency Matrix vs Adjacency Lists

- Adjacency Matrix: $O(n^2)$
- Adjacency Lists : $O(n+m)$
- If the graph is sparse (has few edges)
  - $m \ll n^2$
  - hence Adj Lists based algorithms may be more efficient than Adj Matrix based algorithms
- If the graph is dense (has many edges)
  - $m \approx n^2/2$ (for unigraph) or $m \approx n^2$ (for digraph)
  - Adj Matrix based algorithms is more efficient than Adj List based algorithms

Weighted Graphs

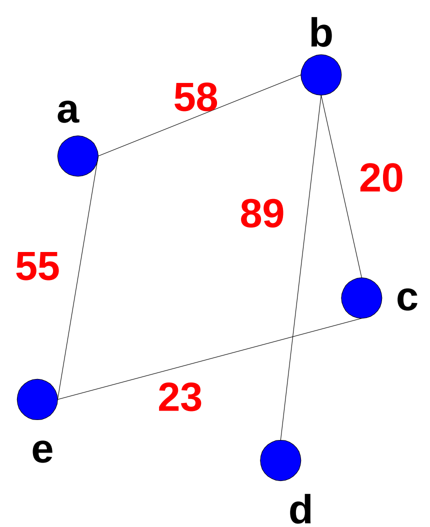
- **Cost adjacency matrix**
  - ☞ **C(i,j) = cost of edge (i,j)**
- **Adjacency lists => each list element is a pair (adjacent vertex, edge weight)**

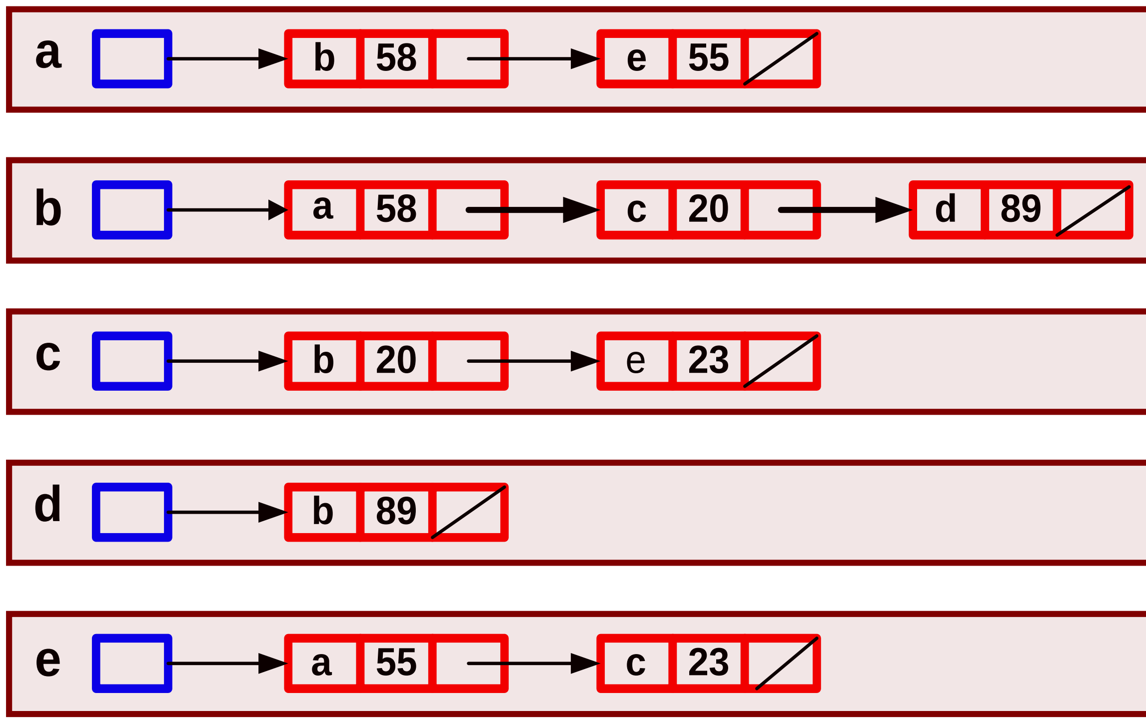


	a	b	c	d	e
a	0	58	0	0	55
b	58	0	20	89	0
c	0	20	0	0	23
d	0	89	0	0	0
e	55	0	23	0	0

Weighted Graphs

- **Cost adjacency matrix.**
  - C(i,j) = cost of edge (i,j)
- **Adjacency lists => each list element is a pair (adjacent vertex, edge weight)**

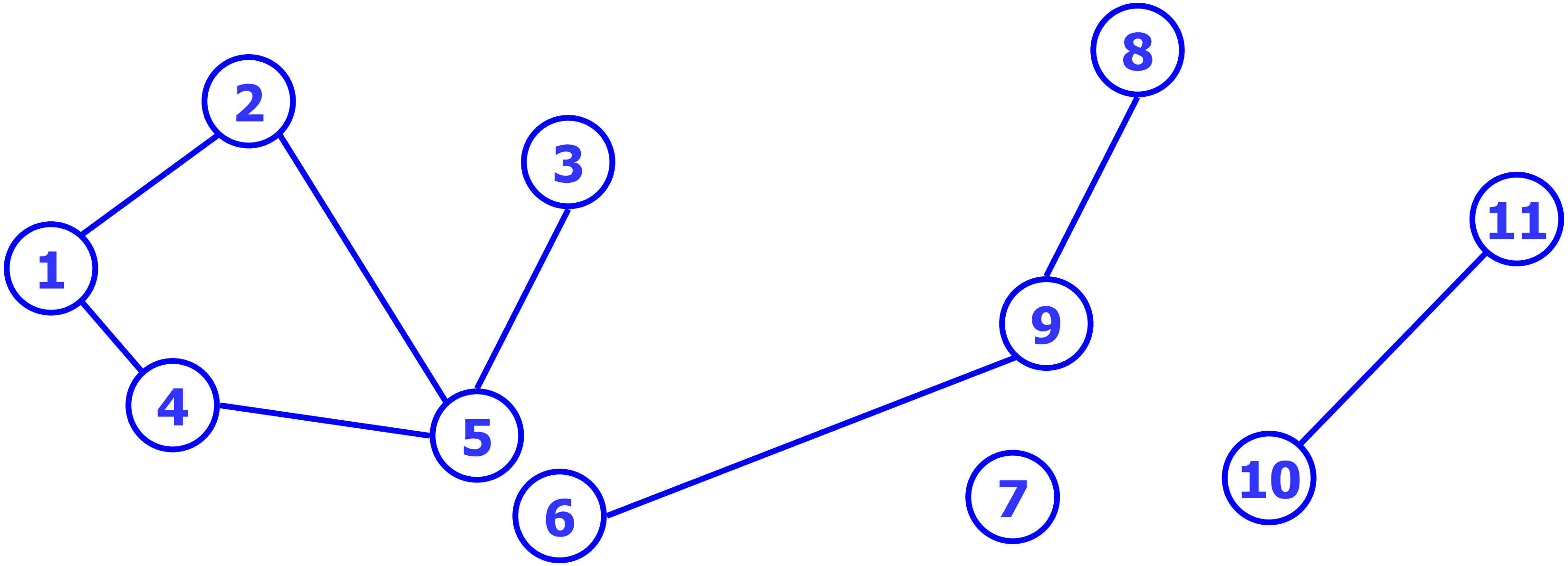




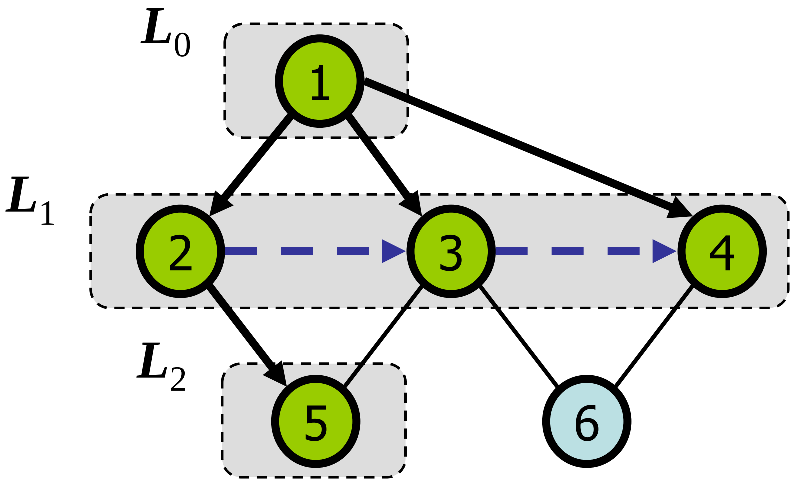
Graph Search Methods

Graph Search Methods

- A vertex u is reachable from vertex v iff there is a path from v to u.
- A search method starts at a given vertex v and visits/labels/marks every vertex that is reachable from v.



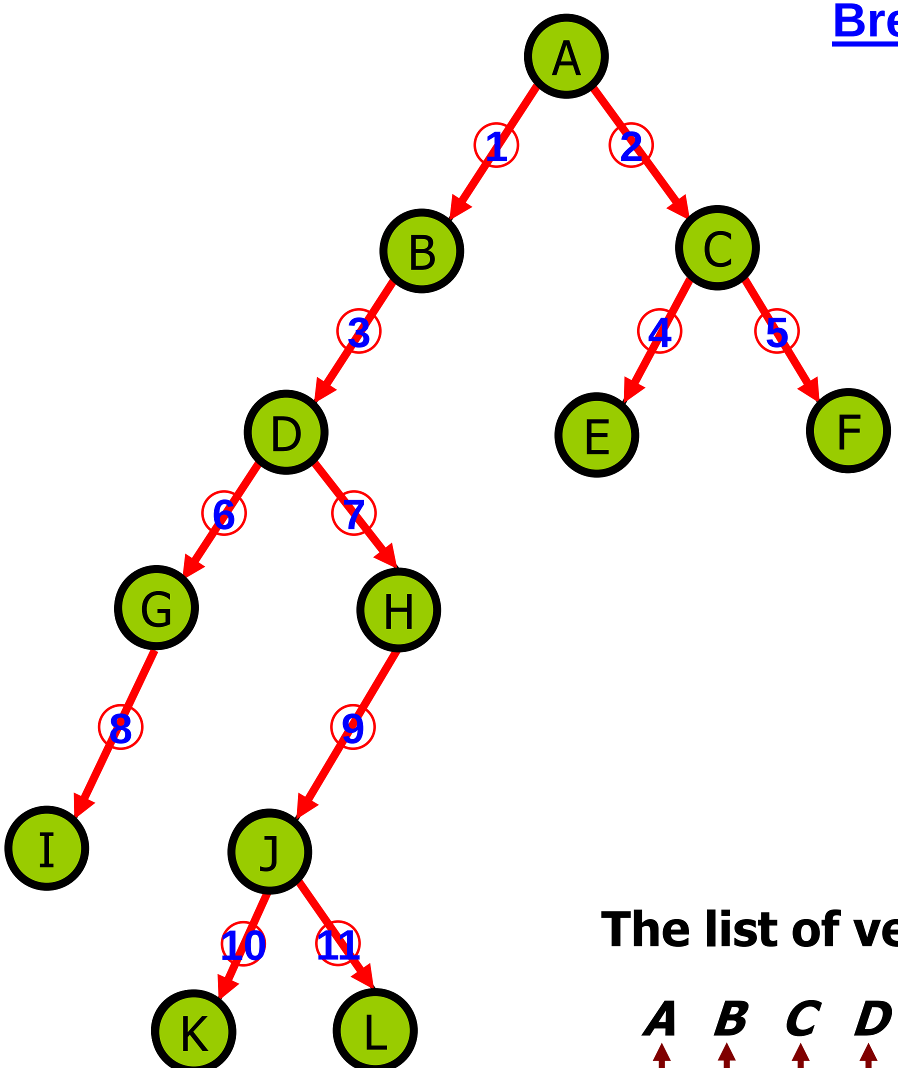
Breadth-First Search



Breadth-First Search

- **Given a source vertex s, explores the edges to “discover” every vertex that is reachable from s.**
- **Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.**
- **Order that vertices are discovered is a “breadth-first tree” that contains all reachable vertices from s.**

Example



Breadth-First-Search (s)

1) visit s, label s as visited.
2) add s to a queue q.
3) while q is not empty
   i) return the front value of q and store it as v
   ii) visit each unvisited vertex u adjacent to v, and add u to the queue q.
   iii) remove the front value of q

The list of vertices visited in order is:

A B C D E F G H I J K L

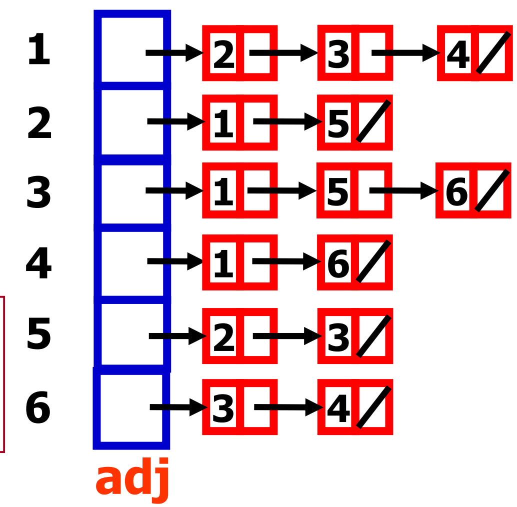
Breadth-First Search

```
bfs (adj, s)  {
    n = adj.last
    for i = 1 to n
        visit[i] = false
    // visit s
    visit[s] = true
    q.enqueue(s)
    while (!q.empty()) {
        v = q.front()
        ref = adj[v]
        while (ref != null) {
            if (!visit[ref.data]) {
                // visit ref.data
                visit[ref.data] = true
                q.enqueue(ref.data)
            }
            ref = ref.next
        }
        q.dequeue()
    }
}
```

This is a template: depending on application, do something here!

<u>**Breadth-First-Search (s)**</u>

1) **visit s, label s as visited.**
2) **add s to a queue q.**
3) **while q is not empty**
    - i) **return the front value of q and store it as v**
    - ii) **visit each unvisited vertex u adjacent to v, and add u to the queue q.**
    - iii) **remove the front value of q**



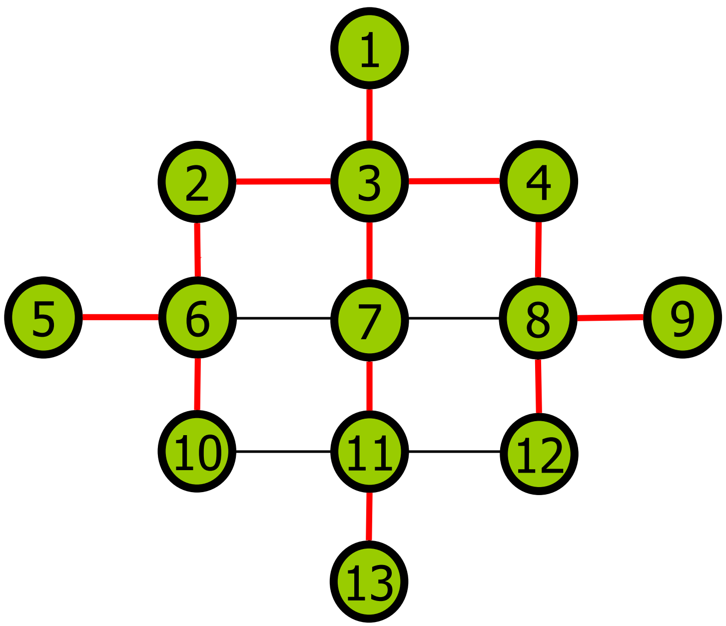
Breadth-First Search

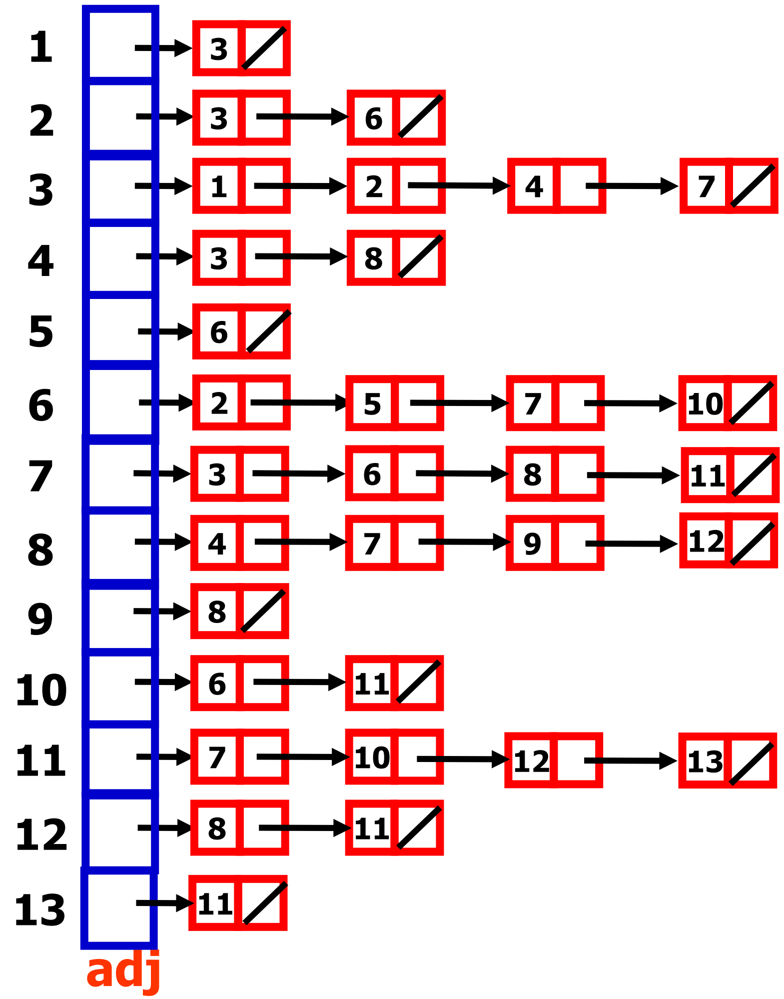
- Visit start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Breadth-First Search

- This algorithm executes a breadth-first search beginning at vertex *s*
- The graph is represented using adjacency lists
  - ☞ *adj*[*i*] is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex *i*
- To track visited vertices, the algorithm uses an array *visit*
  - ☞ *visit*[*i*] is set to true if vertex *i* has been visited or to false if vertex *i* has not been visited.

Example





The list of vertices visited in order is:

1, 3, 2, 4, 7, 6, 8, 11, 5, 10, 9, 12, 13

Time Complexity of Breadth-First Search

```
bfs (adj, s)  {
    n = adj.last
    for i = 1 to n
        visit[i] = false
    // visit s
    visit[s] = true
    q.enqueue(s)
    while (!q.empty()) {
        v = q.front()
        ref = adj[v]
        while (ref != null) {
            if (!visit[ref.data]) {
                // visit ref.data
                visit[ref.data] = true
                q.enqueue(ref.data)
            }
            ref = ref.next
        }
        q.dequeue()
    }
}
```

O(n)

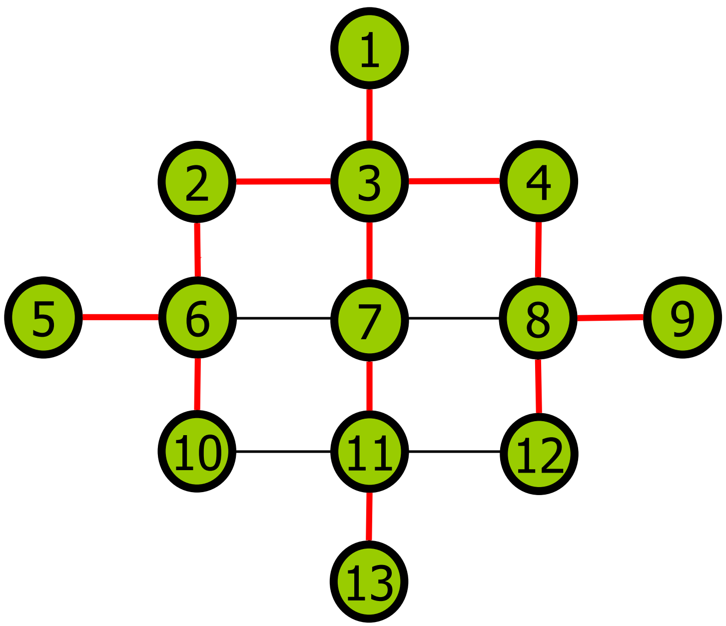
Visit nodes in adjacency lists

in the worst case, each node in adjacency lists is visited once (there are 2m nodes in adj list)

hence time complexity of nested while loops = O(m)

overall time complexity = O(n+m)

Finding Shortest Path Lengths Using BFS



The list of vertices visited in order is:

1, 3, 2, 4, 7, 6, 8, 11, 5, 10, 9, 12, 13

1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞
10	∞
11	∞
12	∞
13	∞

1	0
2	2
3	1
4	2
5	4
6	3
7	2
8	3
9	4
10	4
11	3
12	4
13	4

`length[u] = 1+ length[v]`

Finding Shortest Path Lengths Using BFS

```
bfs (adj, s)  {
    n = adj.last
    for i = 1 to n
        length[i] = ∞ // set as a very large number
    length[s] = 0
    q.enqueue(s)
    while (!q.empty()) {
        v = q.front()
        ref = adj[v]
        while (ref != null) {
            if (length[ref.data]==∞) {
                length[ref.data] = 1+ length[v]
                q.enqueue(ref.data)
            }
            ref = ref.next
        }
        q.dequeue()
    }
}
```

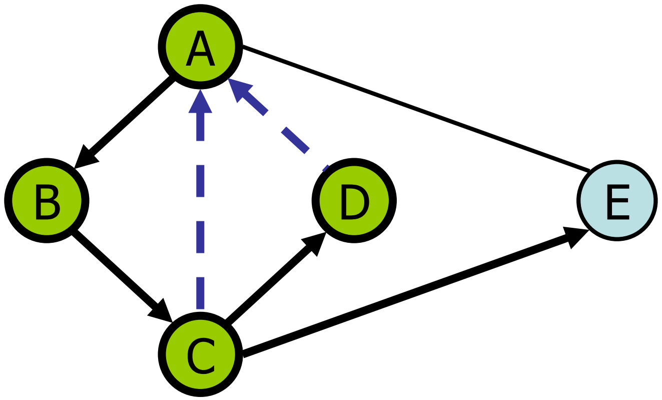
This algorithm finds the length of a shortest path from the start vertex start to every other vertex in a graph with vertices 1, ... , n

length[i] is set to the length of a shortest path from *start* to vertex i

Learning Takeaway

- **Using the *template* method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time**
  - ☞ **Compute the connected components of G**
  - ☞ **Compute a spanning forest of G**
  - ☞ **Find a simple cycle in G, or report that G is a forest**
  - ☞ **Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists**

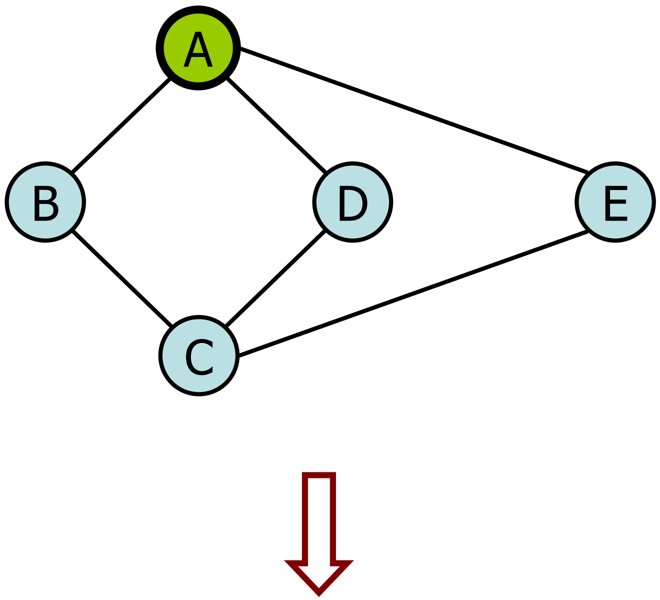
Depth-First Search



Depth-First Search

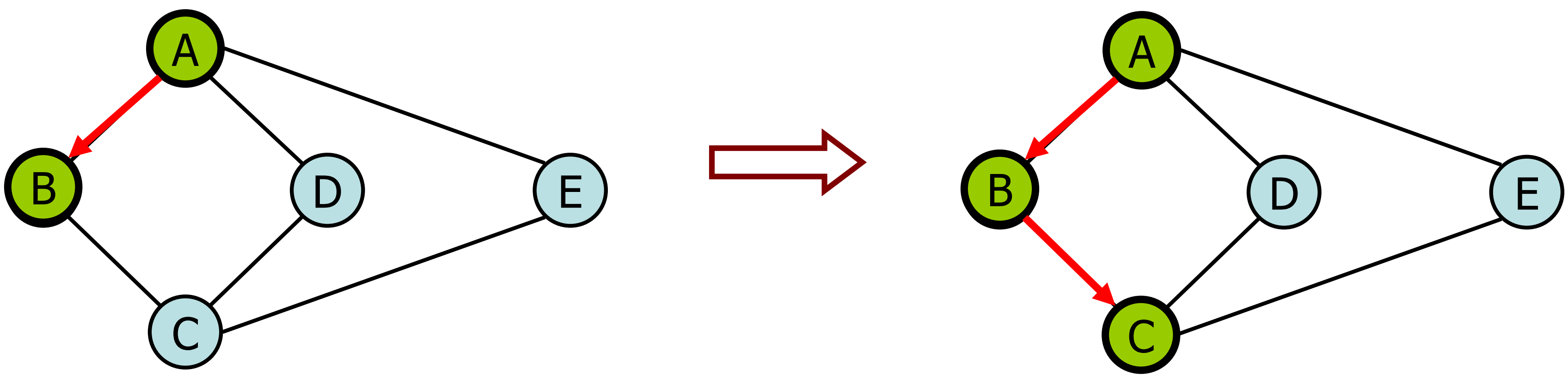
- **Search "deeper" in the graph whenever possible**
- **Explores edges out of the most recently visited vertex *v* that still has unvisited neighbors.**
- **If all of *v*'s neighbors have been visited, "backtracks" to vertex from which *v* was visited.**
- **Continue process from there until we have visited all vertices reachable from original first vertex.**

Example



<u>Depth-First-Search (v)</u>

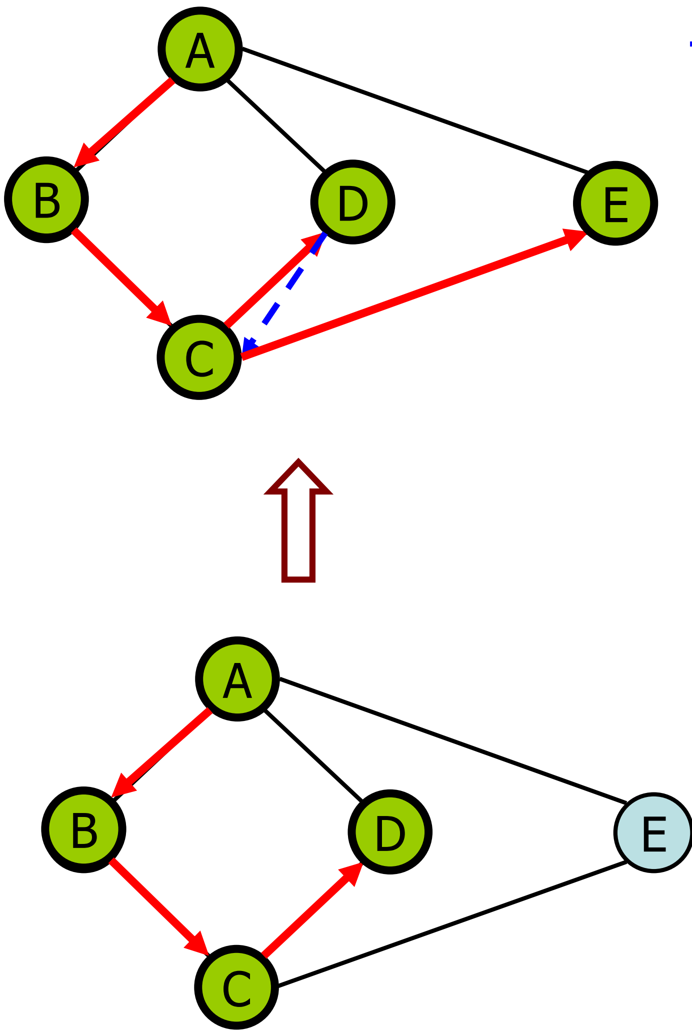
1) **visit v, label v as visited.**
2) **For each unvisited vertex u adjacent to v, execute Depth-First-Search on u.**

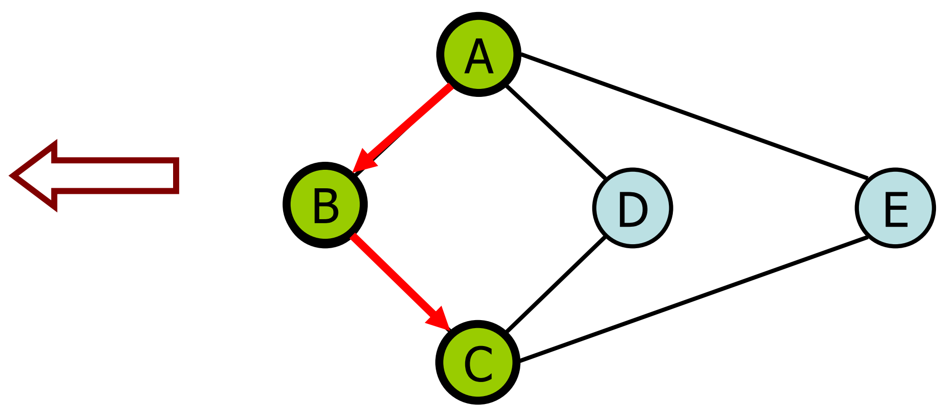


Example

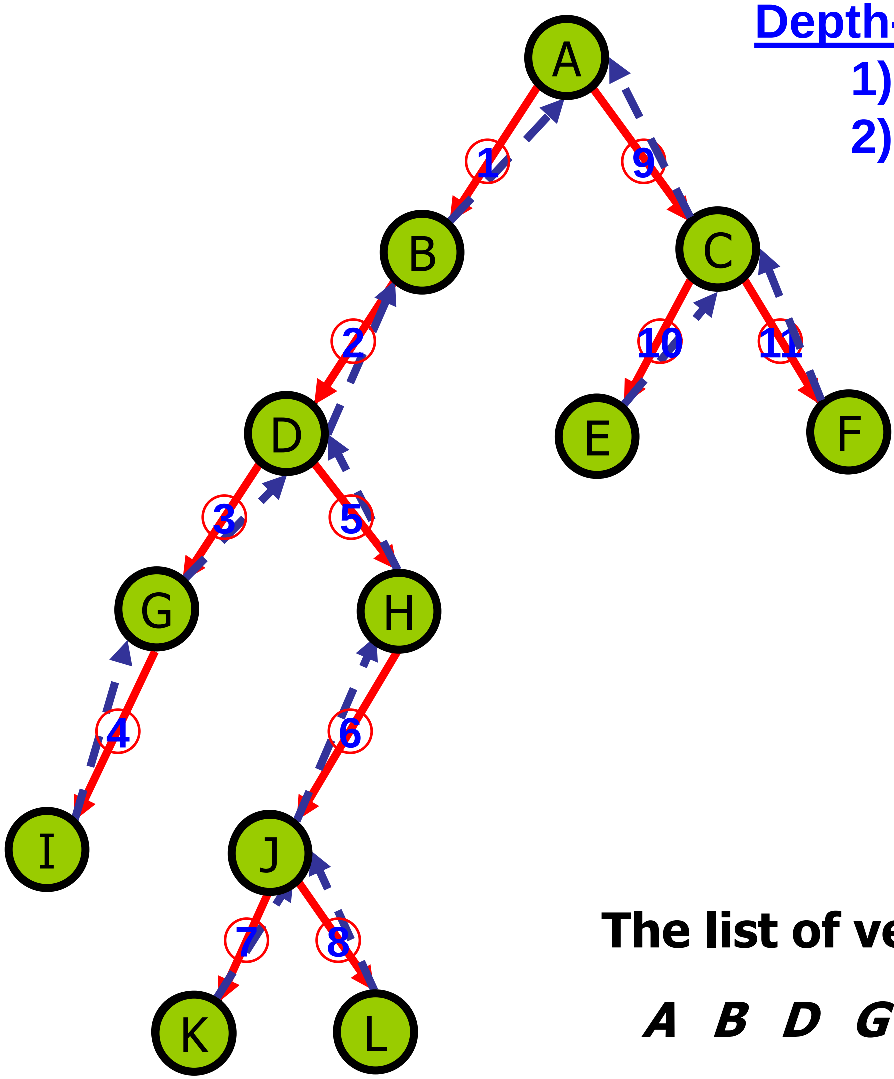
<u>Depth-First-Search (v)</u>

1) **visit v, label v as visited.**
2) **For each unvisited vertex u adjacent to v, execute Depth-First-Search on u.**





Example



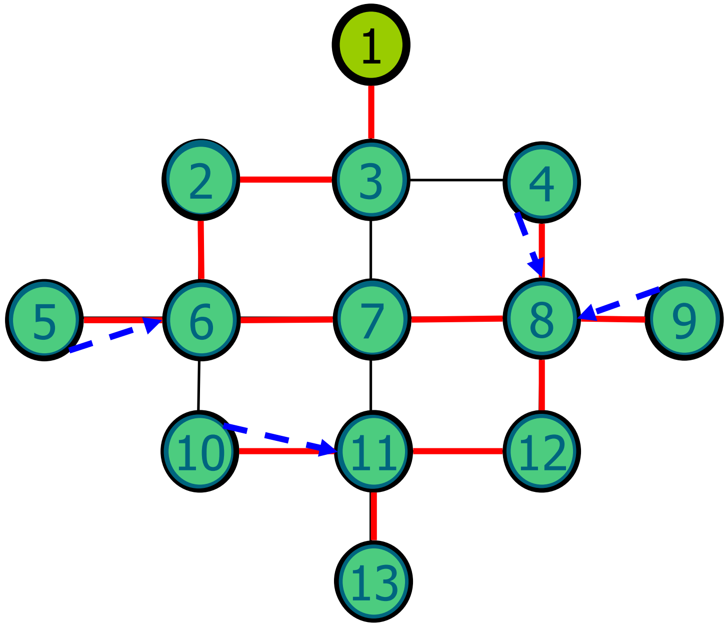
Depth-First-Search (v)

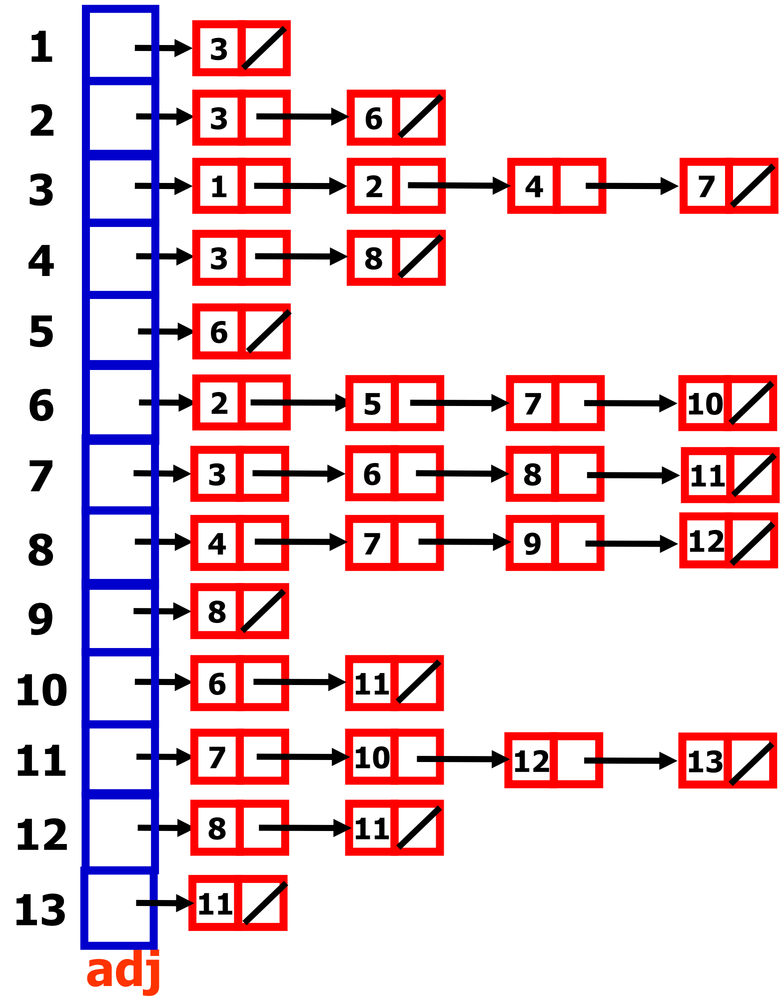
1) visit v, label v as visited.
2) For each unvisited vertex u adjacent to v, execute Depth-First-Search on u.

The list of vertices visited in order is:

A B D G I H J K L C E F

Depth-First Search





The list of vertices visited in order is:

1, 3, 2, 6, 5, 7, 8, 4, 9, 12, 11, 10, 13

Depth-First Search

```
dfs(adj, s)  {
    n = adj.last
    for (i = 1 to n) {
        visit[i] = false
    }
    dfs_recurs(adj,visit,s)
}

dfs_recurs(adj,visit,v) {
    visit[v] = true
    ref = adj[v]
    while (ref != null) {
        if (!visit[ref.data])
            dfs_recurs(adj,visit,ref.data)
        ref = ref.next
    }
}
```

Depth-First-Search (v)

1) visit v, label v as visited.
2) For each unvisited vertex u adjacent to v, execute Depth-First-Search on u.

Depth-First Search

- The graph is represented using adjacency lists
  - *adj*[*i*] is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex *i*.
- To track visited vertices, the algorithm uses an array *visit*
  - *visit[i]* is set to true if vertex *i* has been visited or to false if vertex *i* has not been visited.

```
dfs(adj, s)  {
    n = adj.last
    for (i = 1 to n) {
        visit[i] = false
    }
    dfs_recurs(adj,visit,s)
}

dfs_recurs(adj,visit,v) {
    visit[v] = true
    ref = adj[v]
    while (ref!= null) {
      if (!visit[ref.data])
        dfs_recurs(adj,visit,ref.data)
      ref = ref.next
    }
}
```

DFS Time Complexity

```
dfs(adj, s)  {
    n = adj.last
    for (i = 1 to n) {
        visit[i] = false
    }
    dfs_recurs(adj,visit,s)
}

dfs_recurs(adj,visit,v) {
    visit[v] = true
    ref = adj[v]
    while (ref!= null) {
        if (!visit[ref.data])
            dfs_recurs(adj,visit,ref.data)
        ref = ref.next
    }
}
```

$O(n) \le k_1 n$

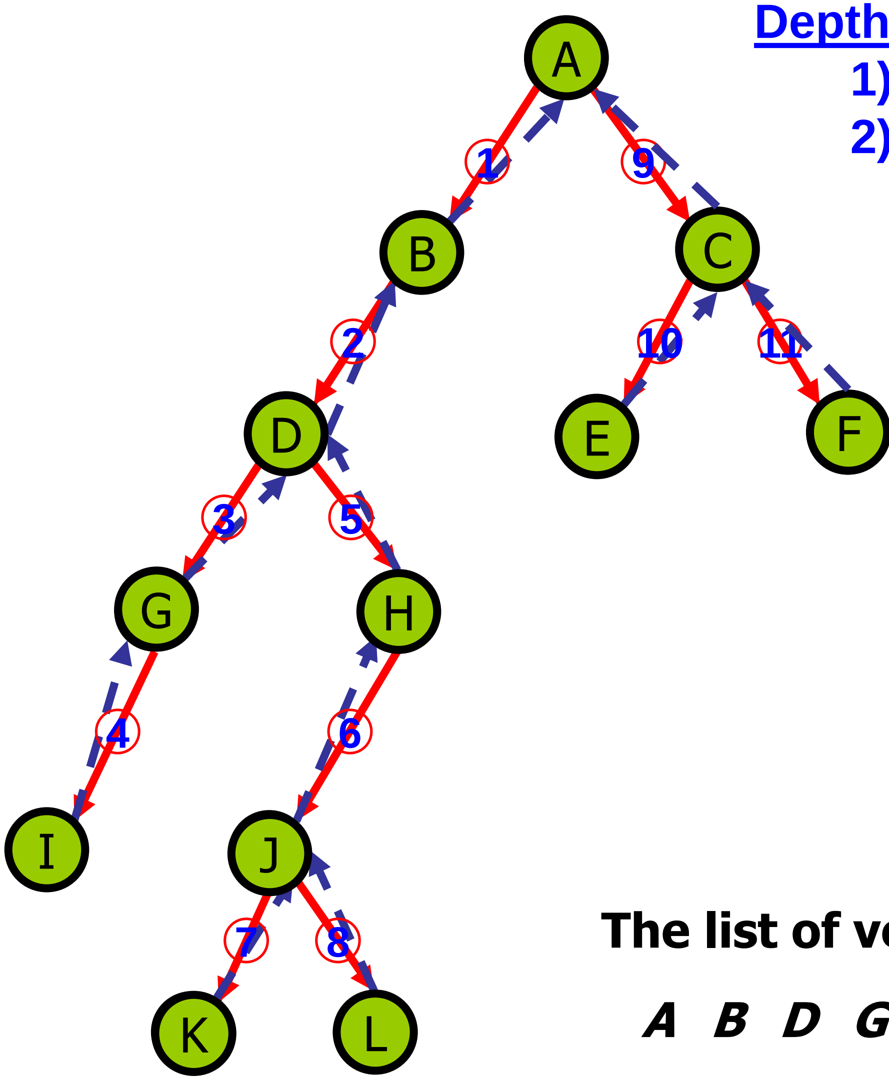
Visit nodes in adjacency lists

$O(m) \le k_2 m$

$$O(n) + O(m) \le k_1 n + k_2 m$$
$$\le max(k_1, k_2)(n + m)$$

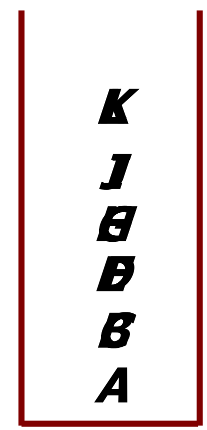
Overall time complexity = $O(n + m)$

Stack Implementation



<u>Depth-First-Search (v)</u>

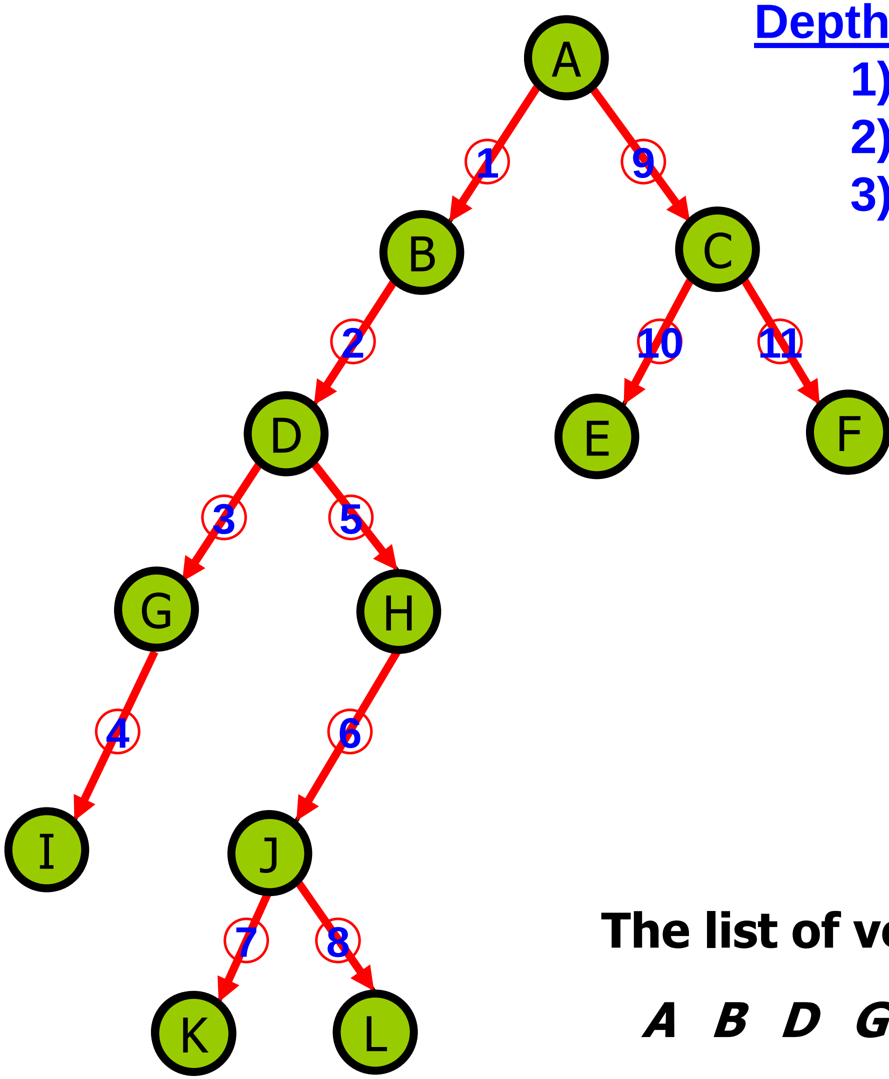
1) visit v, label v as visited.
2) For each unvisited vertex u adjacent to v, execute Depth-First-Search on u.



The list of vertices visited in order is:

A B D G I H J K L C E F

Stack Implementation



Depth-First-Search (start)

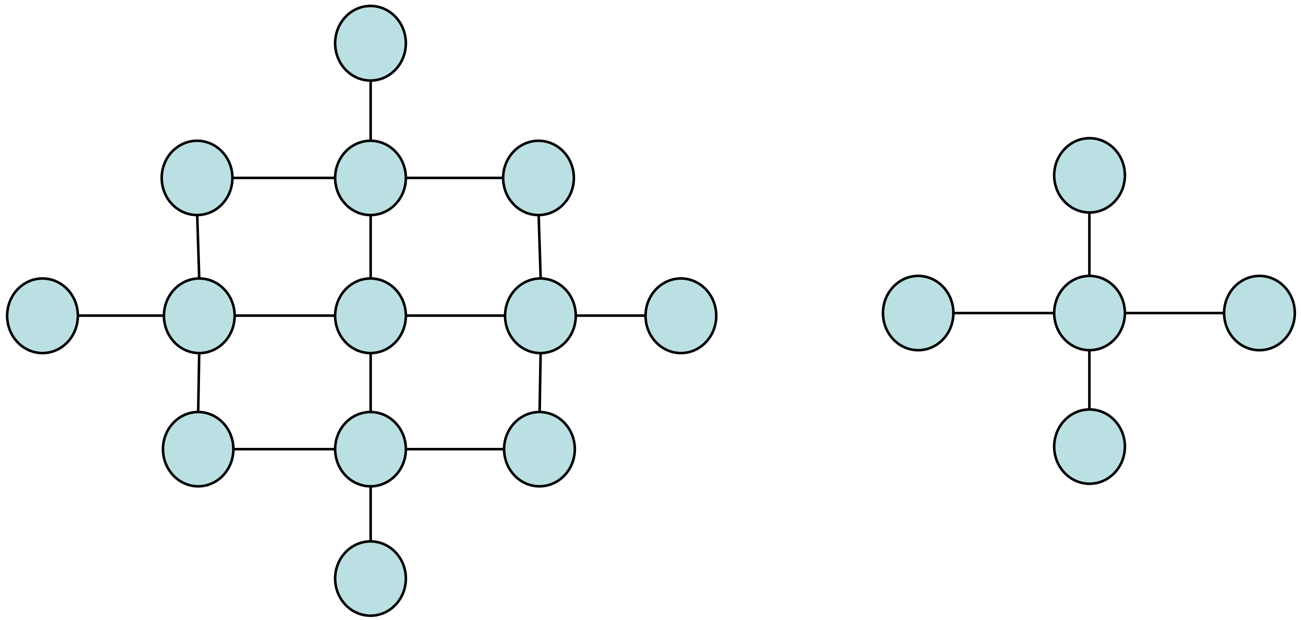
1) visit start, label start as visited.
2) push start to a stack s.
3) while s is not empty
   i) return the top value of s and store it as v
   ii) If v has one unvisited adjacent vertex u, visit u and push u to the stack s.
   iii) If v does not have unvisited adjacent vertices, remove the top value v of s

The list of vertices visited in order is:

A B D G I H J K L C E F

An Application of DFS

- DFS can be used to test whether a graph is connected.
  - Run DFS using any vertex as the start vertex
  - Upon completing of the algorithm, check whether all vertices are visited
  - The graph is connected if and only if all vertices are visited, i.e. all vertices are reachable from the start vertex



Using DFS to search whole graph

- Perform DFS on any starting vertex.
- If any unvisited vertices remain, select one of them as new source and repeats search.
- Algorithm ends only when every vertex has been visited.

```
dfs(adj)  {
    n = adj.last
    for (i = 1 to n) {
        visit[i] = false
    }
    for (i = 1 to n) {
        if (!visit[i])
            dfs_recurs(adj,visit,i)
    }
}
```

Learning Takeaway

- DFS is another graph search method, different from BFS.

- Depending on the application, either DFS or BFS or both can be used.

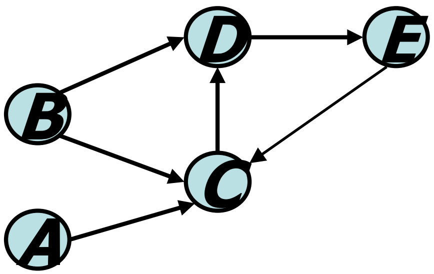
- Application examples in which DFS can be used:
  - ☞ compute the connected components of G
  - ☞ compute a spanning forest of G
  - ☞ find a simple cycle in G, or report that G is a forest
  - ☞ topological sorting (we will study this next)

Topological Sorting

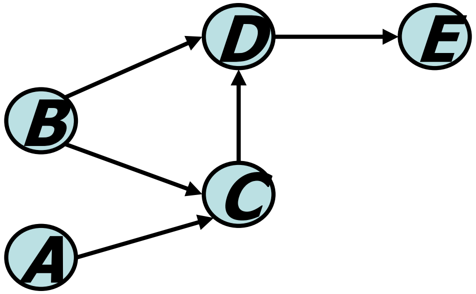
Topological Sort

- A directed acyclic graph (DAG) is a digraph that has no directed cycles

G1



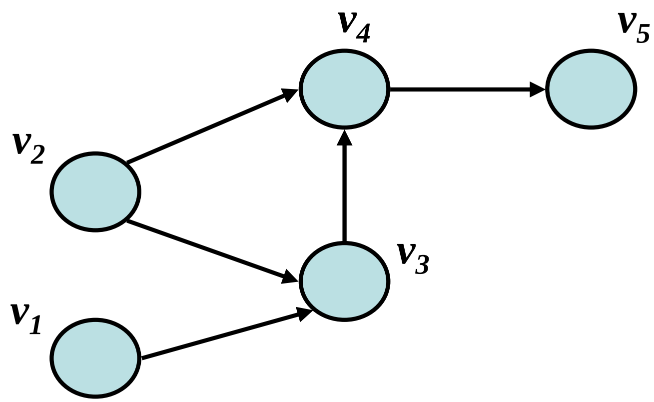
Is G1 a DAG ?



G2

Is G2 a DAG ?

Topological Sort



Topological sort of G: v_2, v_1, v_3, v_4, v_5

- **A topological sorting of a DAG is an ordering of the vertices such that in that list, v_i precedes v_j whenever a path exists from v_i to v_j**

Topological Sort

- **We will discuss:**
  - ☞ **the idea of sorting elements in a DAG**
    - ✓ **examples of topological sort**
  - ☞ **the implementation**
    - ✓ **using a table of in-degrees**
    - ✓ **using DFS**

Topological Sort

- **Given two vertices v_i and v_j in a DAG, at most, there can exist only:**
  - **a path from v_i to v_j, or**
  - **a path from v_j to v_i**

Proof:

Assume otherwise, there exists two paths:

$$(v_i, v_{1,1}, v_{1,2}, v_{1,3}, \ldots, v_j)$$

$$(v_j, v_{2,1}, v_{2,2}, v_{2,3}, \ldots, v_i)$$

Thus, $(v_i, v_{1,1}, v_{1,2}, v_{1,3}, \ldots, v_j, v_{2,1}, v_{2,2}, v_{2,3}, \ldots, v_i)$ is a path which is also a cycle: contradiction

Topological Sort

- **Thus, it must be possible to list all of the vertices such that in that list, v_i precedes v_j whenever a path exists from v_i to v_j.**

- **If this is not possible, this would imply the existence of a cycle.**

Topological Sort

Theorem:

A graph is a DAG if and only if it has a topological sorting.

Proof:

Such a statement is of the form $a \leftrightarrow b$ and this is equivalent to:

$$a \rightarrow b \text{ and } b \rightarrow a$$

How do we go around proving this? To prove $a \rightarrow b$ we may:

Assume a is true and then show b must also be true: $a \rightarrow b$

Assume b is false and then show a must also be false: $\neg b \rightarrow \neg a$

Topological Sort

We will start with showing $a \rightarrow b$:

if a graph is a DAG, it has a topological ordering

By induction:

A graph with one vertex 1 is a DAG and it has a topological sort

Assume a DAG with n vertices has a topological sort

A DAG with $n + 1$ vertices must have at least one vertex v of in-degree zero, otherwise, one could always follow such edges back until one ultimately created a cycle

- This contradicts the assumption that we have a DAG

Removing the vertex v and its edges creates a graph with n vertices

- If this sub-graph has a cycle, the original graph would also have a cycle—contradiction
- Thus, the graph with n vertices is also a DAG, therefore it has a topological ordering

Add the vertex v to the start of the topological ordering to get one for the graph of size $n + 1$

Topological Sort

Next, we will show that $b \rightarrow a$:

if a graph has a topological ordering, it must be a DAG

We will show this by showing $\neg a \rightarrow \neg b$:

Assume that a graph is not a DAG and show it cannot have a topological sort

Therefore, it has a cycle: $(v_1, v_2, v_3, \ldots, v_k, v_1)$

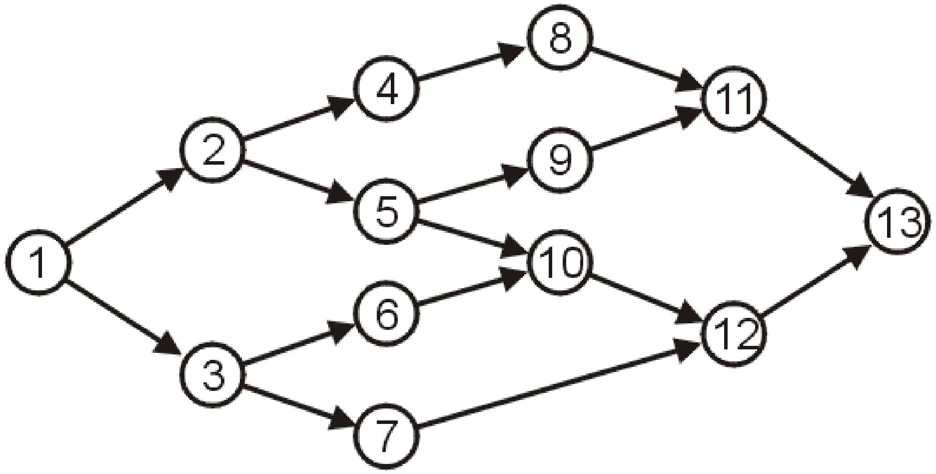
- ✓ In any topological sort, v_1 must appear before v_2, because there exists a path (v_1, v_2)
- ✓ However, there is also a path from v_2 to v_1: $(v_2, v_3, \ldots, v_k, v_1)$
- ✓ Therefore, v_2 must appear in the topological sort before v_1

This is a contradiction, therefore the graph cannot have a topological sort.

The theorem is now proved.

Topological Sort

- For example, in this DAG, one topological sort is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

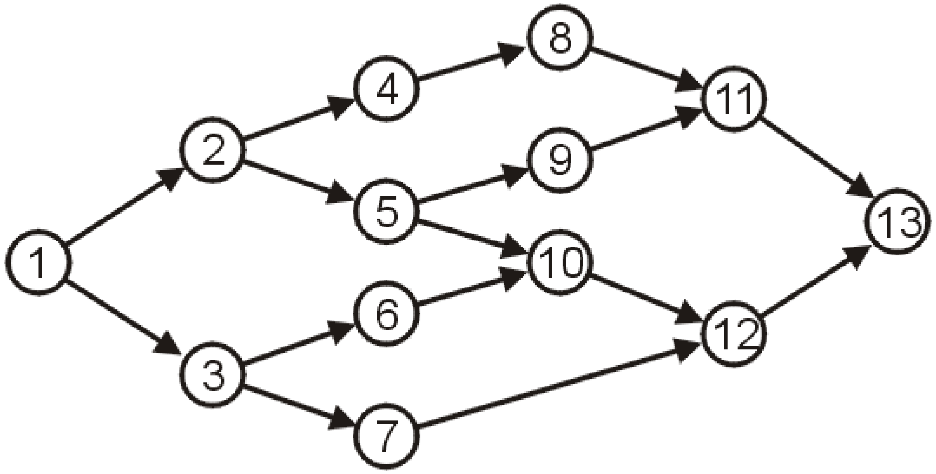


Topological Sort

A topological sort may not be unique

Two further topological sorts are:

- **1, 3, 2, 7, 6, 5, 4, 10, 9, 8, 12, 11, 13**
- **1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13**

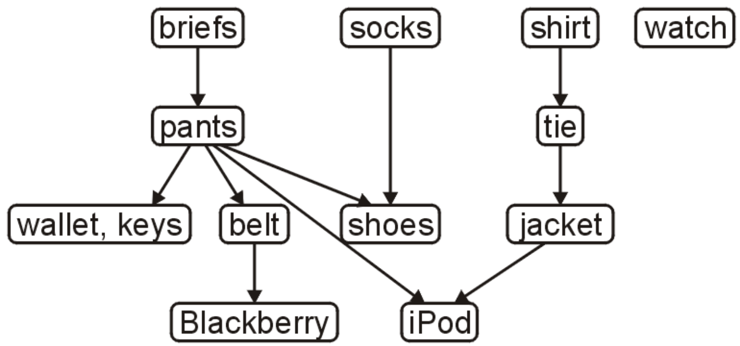


Application

- **Given a number of tasks, there are often a number of constraints between the tasks:**
  - ☞ **task A must be completed before task B can start**
- **These tasks together with the constraints form a directed acyclic graph**
- **A topological sort of the graph gives an order in which the tasks can be scheduled while still satisfying the constraints**

Application

The following is a task graph for getting dressed:



One topological sort is:

briefs, pants, wallet, keys, belt, Blackberry, socks, shoes, shirt, tie, jacket, iPod, watch

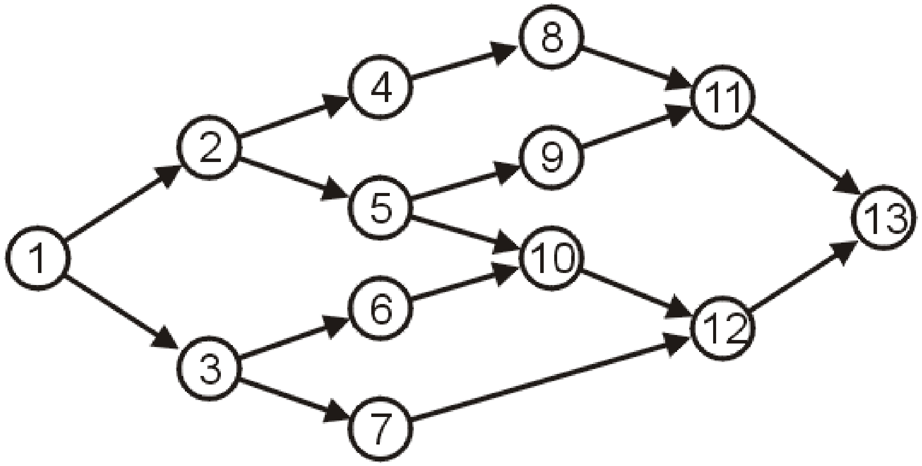
A more likely topological sort is:

briefs, socks, pants, shirt, belt, tie, jacket, wallet, keys, Blackberry, iPod, watch, shoes

Topological Sort

- **To generate a topological sort, we note that we must start with a vertex with an in-degree of zero:**

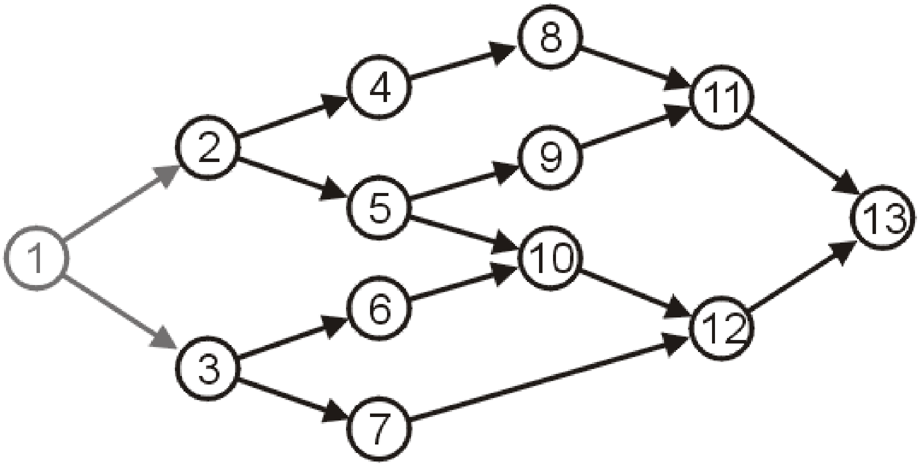
  1



Topological Sort

- At this point, we may consider those edges which connect vertex **1** to other vertices, and thus, we may chose vertex **2**:

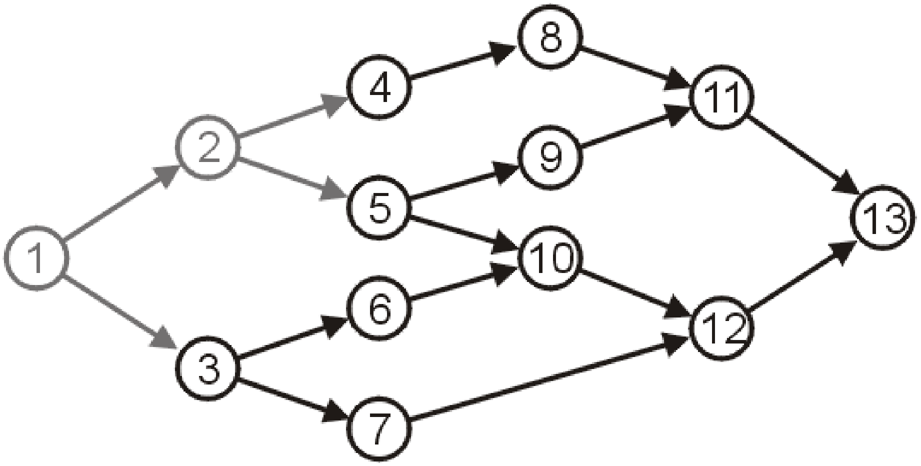
  1, 2



Topological Sort

- We may now consider edges which extend from either vertices 1 or 2
  We may choose from vertices 4, 5, or 3:

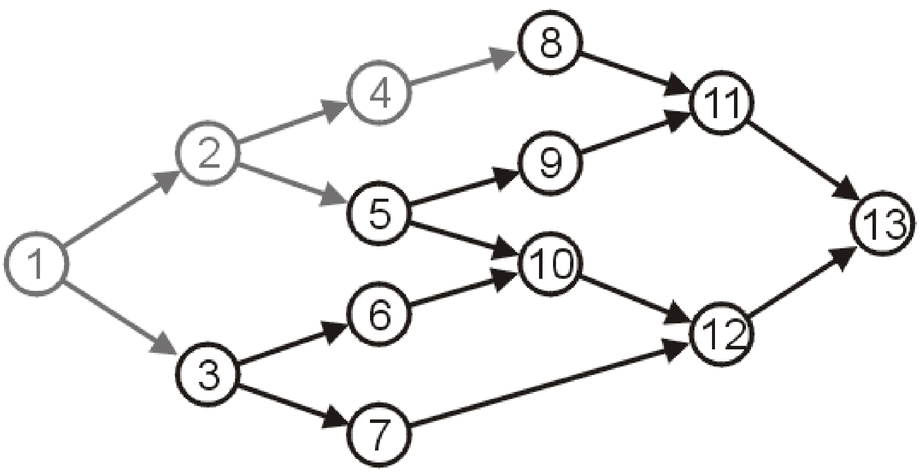
  1, 2, 4



Topological Sort

- Adding the edges extending from vertex 4, we find that we may add vertex 8 to our topological sort:

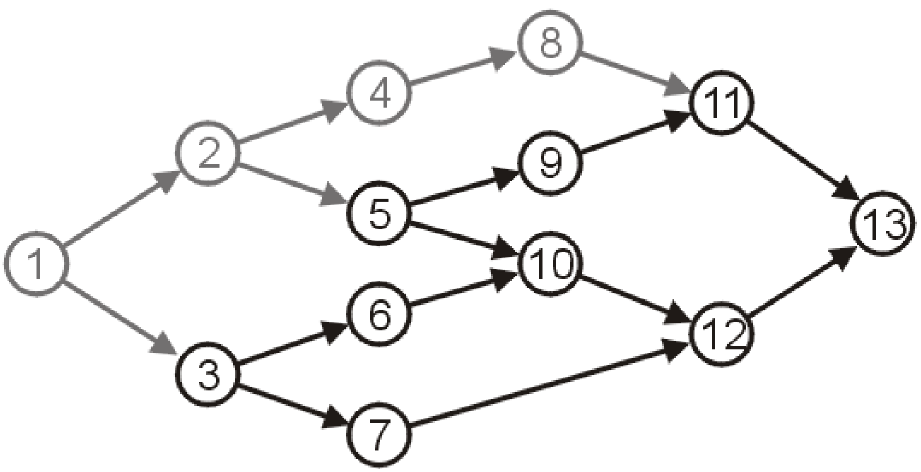
  1, 2, 4, 8



Topological Sort

- At this point, we cannot add 11, as it must follow 9 in the topological sort
  Instead, we must choose from 5 or 3:

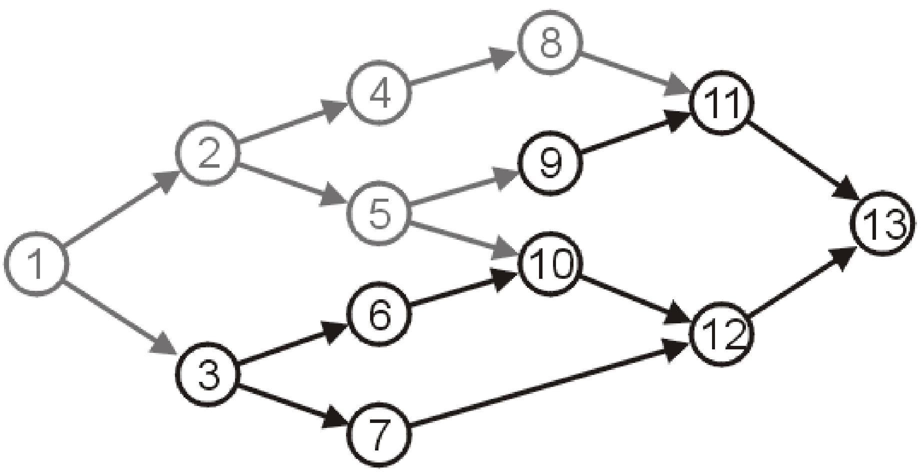
  1, 2, 4, 8, 5



Topological Sort

- Removing vertex 5 from consideration allows us to consider vertices 3 or 9
  We note that 3 must precede 10:

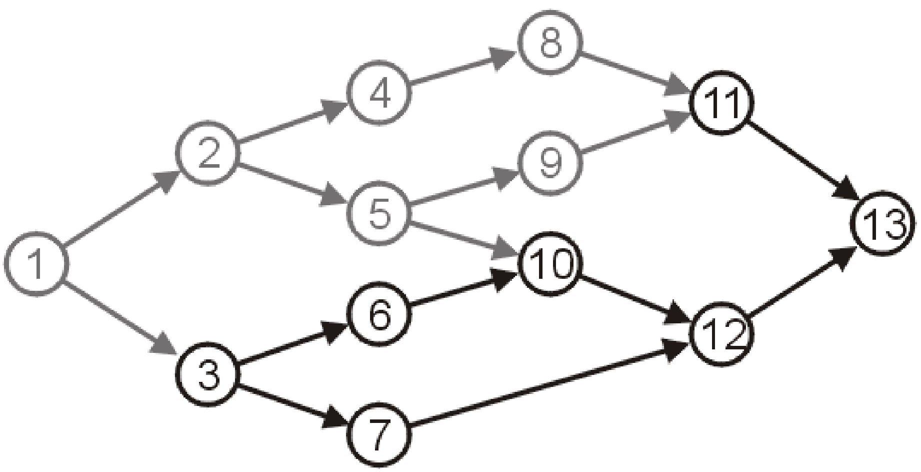
  1, 2, 4, 8, 5, 9



Topological Sort

- We are now free to add 11 to our topological sort

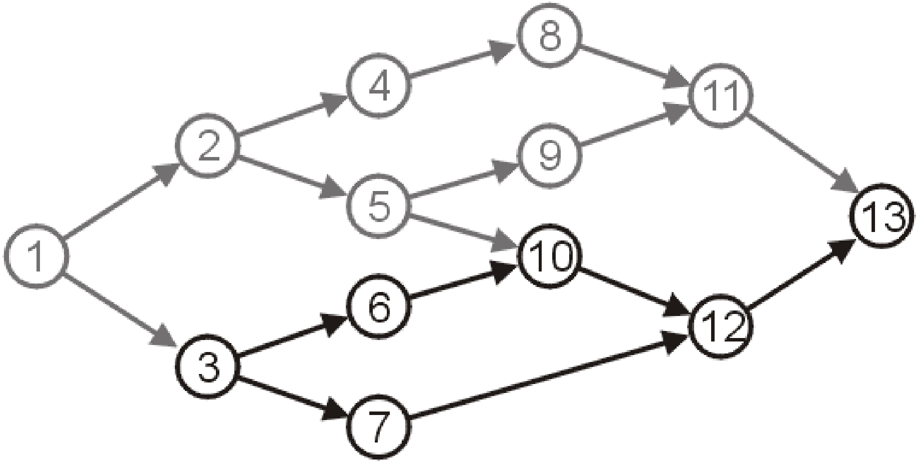
1, 2, 4, 8, 5, 9, 11



Topological Sort

- **The only vertex left which has an in-degree of 0 when we ignore all vertices already in the topological sort is 3**

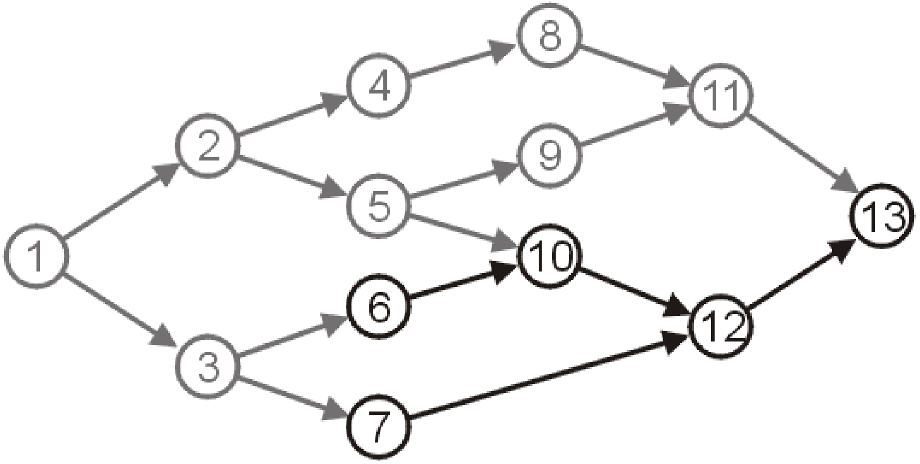
  1, 2, 4, 8, 5, 9, 11, 3



Topological Sort

- Having added 3, this now allows us to choose from either vertices 6 or 7

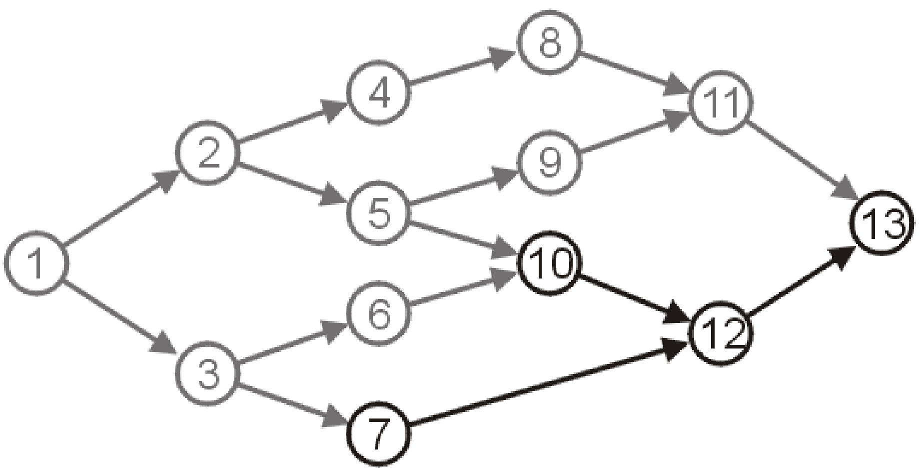
1, 2, 4, 8, 5, 9, 11, 3, 6



Topological Sort

- **Adding vertex 6 to our sort allows us now to choose from vertices 7 or 10**

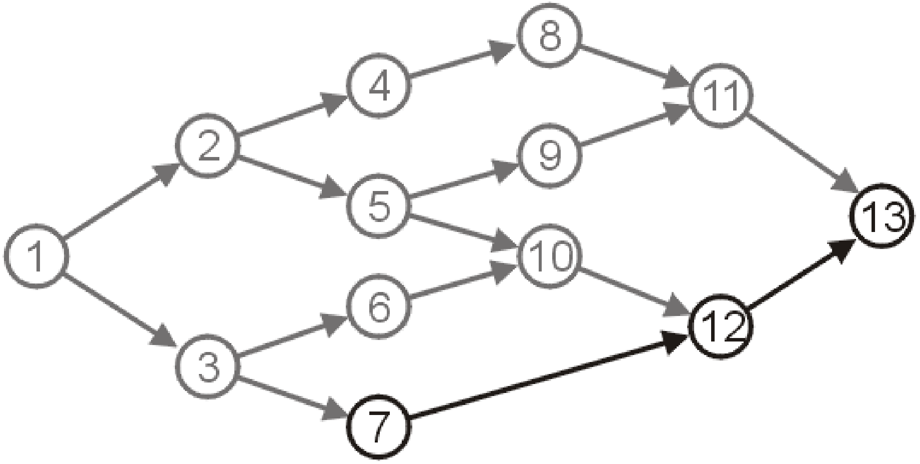
1, 2, 4, 8, 5, 9, 11, 3, 6, 10



Topological Sort

- As 7 must precede 12 in the topological sort, we must now add 7 to the sort

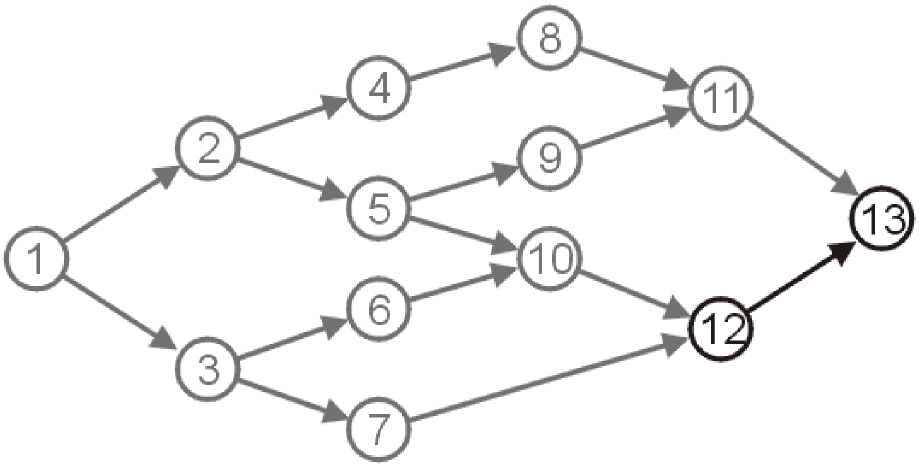
1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7



Topological Sort

- At this point, we add vertex 12

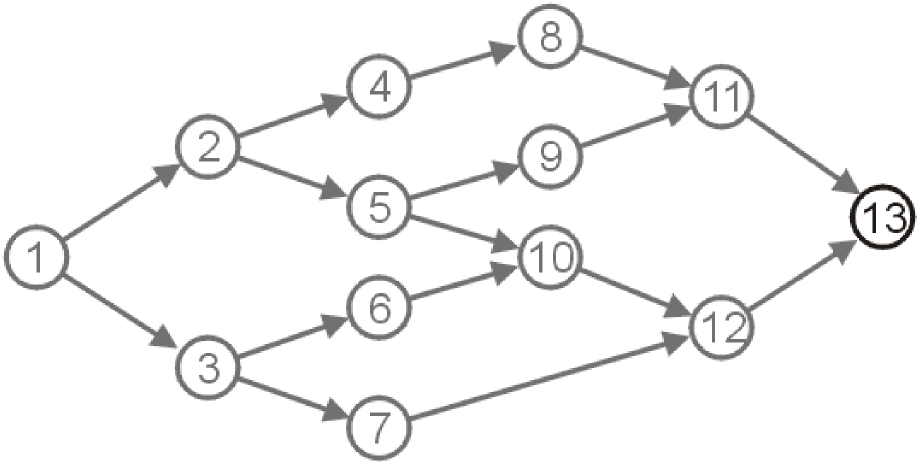
1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12



Topological Sort

- And finally, vertex 13

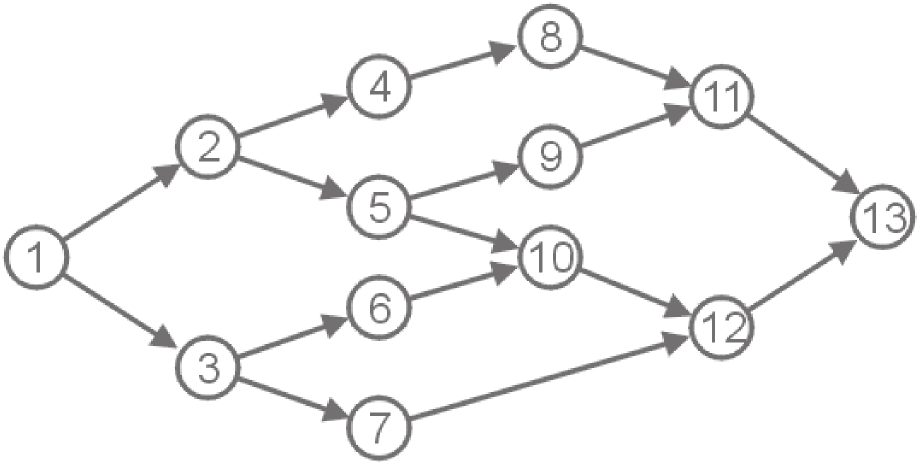
1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13



Topological Sort

- **At this point, there are no vertices left, and therefore we have completed our topological sorting of this graph**

  1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13



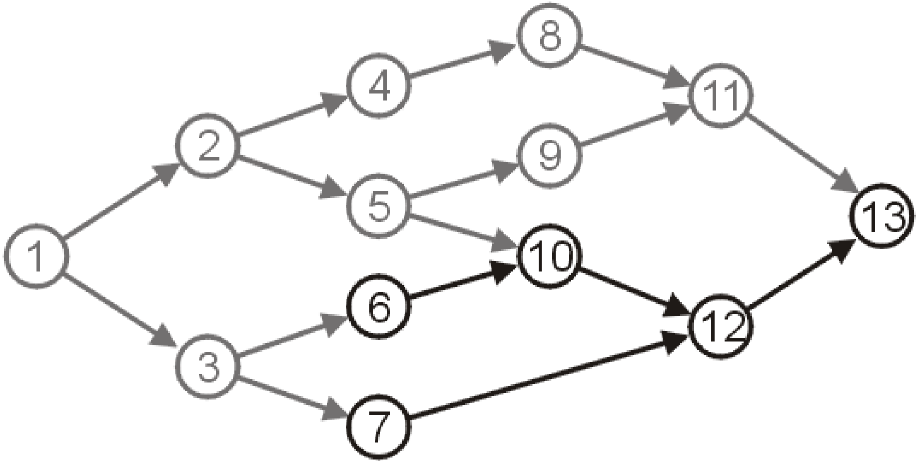
Topological Sort

- It should be obvious from this process that a topological sort is not unique
- At any point where we had a choice as to which vertex we could choose next, we could have formed a different topological sort

Topological Sort

- **For example, at this stage, had we chosen vertex 7 instead of 6:**

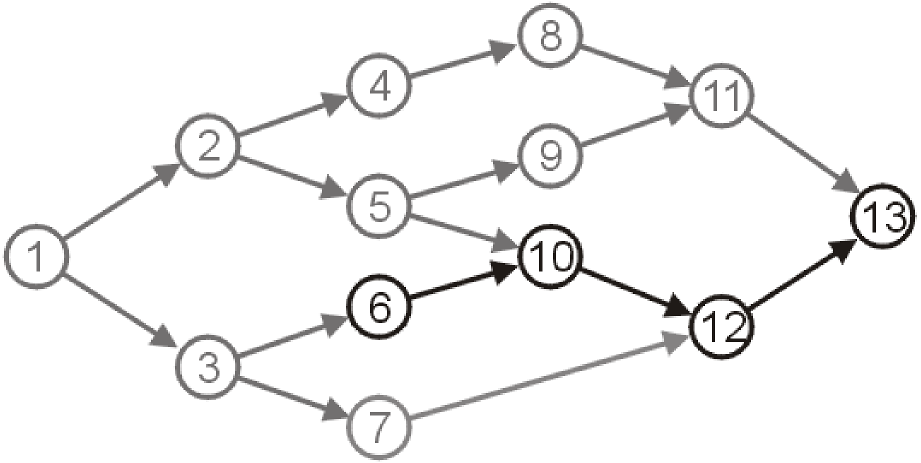
1, 2, 4, 8, 5, 9, 11, 3, 7



Topological Sort

- The resulting topological sort would have been required to be

1, 2, 4, 8, 5, 9, 11, 3, 7, 6, 10, 12, 13



Topological Sort

- Thus, two possible topological sorts are

  1, 2, 4, 8, 5, 9, 11, 3, 6, 10, 7, 12, 13

  1, 2, 4, 8, 5, 9, 11, 3, 7, 6, 10, 12, 13

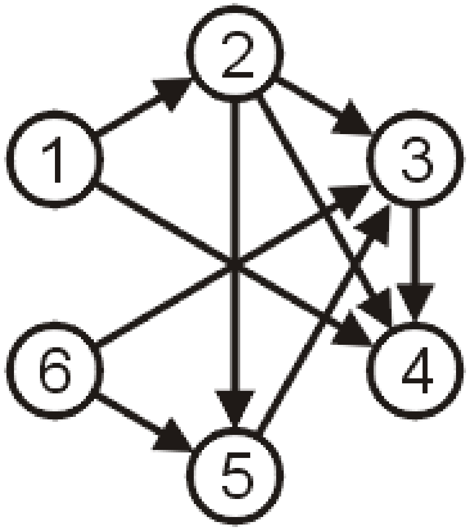
- As seen before, these are not the only topological sorts possible for this graph, for example

  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

  is equally acceptable

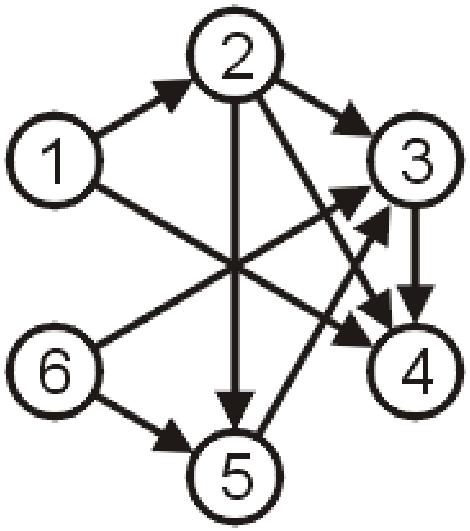
Implementation

- Consider the following DAG with six vertices



Implementation

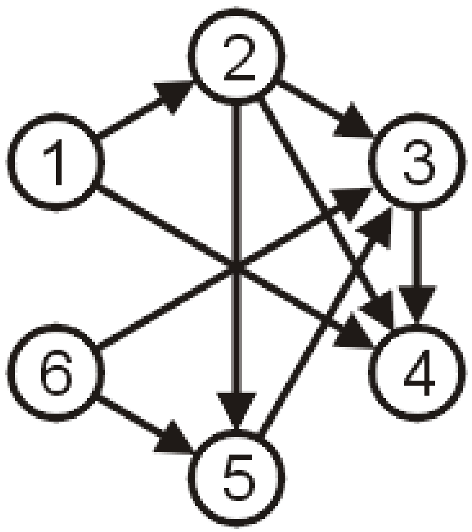
- Let us define the array of in-degrees



1	0
2	1
3	3
4	3
5	2
6	0

Implementation

- **And a queue into which we can insert vertices 1 and 6**



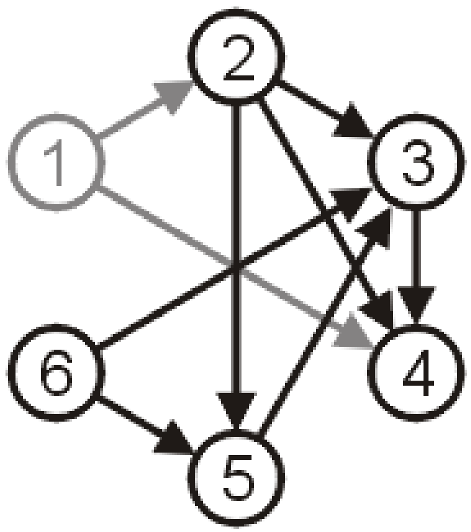
1	0
2	1
3	3
4	3
5	2
6	0

Queue

1	6		

Implementation

- We dequeue the head (1), decrement the in-degree of all adjacent vertices, and enqueue 2



Queue

6	2		

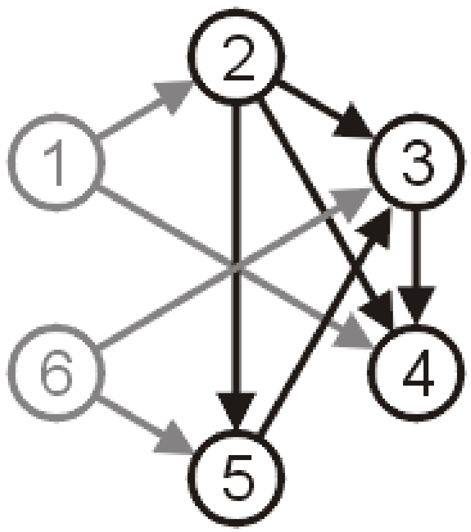
Sort

1

1	0
2	0
3	3
4	2
5	2
6	0

Implementation

- We dequeue 6 and decrement the in-degree of all adjacent vertices



1	0
2	0
3	2
4	2
5	1
6	0

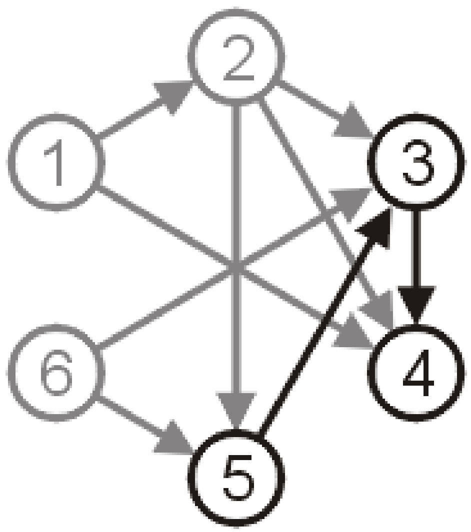
Queue

2			

Sort
1, 6

Implementation

- We dequeue 2, decrement, and enqueue vertex 5



5			

Queue

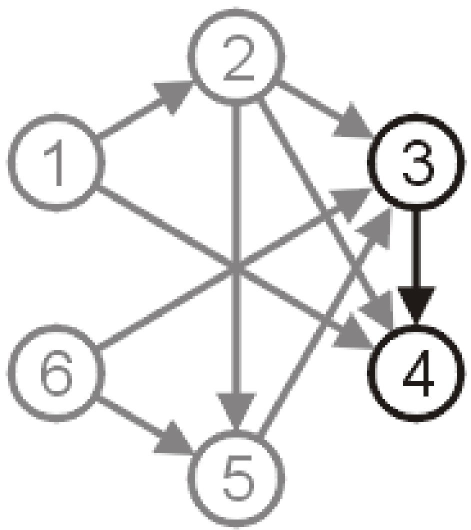
Sort

1, 6, 2

1	0
2	0
3	1
4	1
5	0
6	0

Implementation

- **We dequeue 5, decrement, and enqueue vertex 3**



Queue

3			

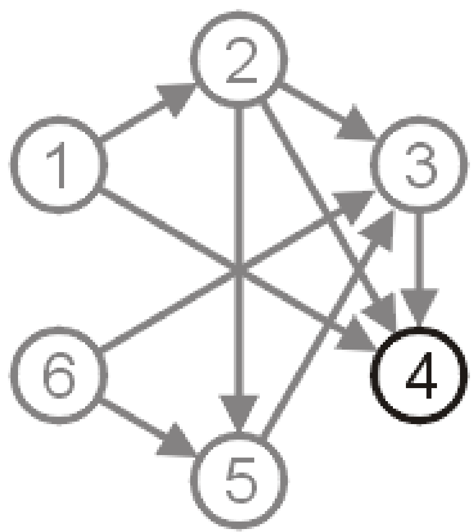
Sort

1, 6, 2, 5

1	0
2	0
3	0
4	1
5	0
6	0

Implementation

- We dequeue 3, decrement 4, and add 4 to the queue



Queue

4			

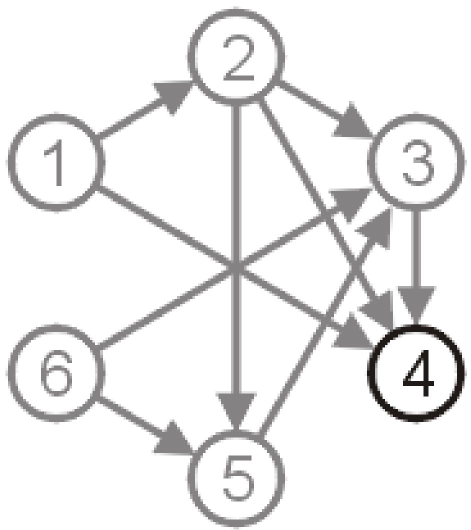
Sort

1, 6, 2, 5, 3

1	0
2	0
3	0
4	0
5	0
6	0

Implementation

- We dequeue 4, there are no adjacent vertices to decrement in degree



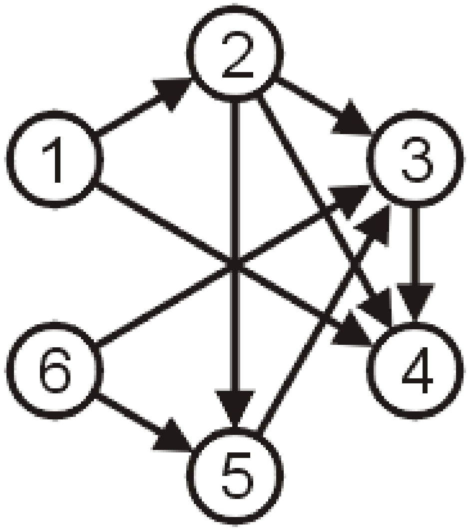
Queue

Sort
1, 6, 2, 5, 3, 4

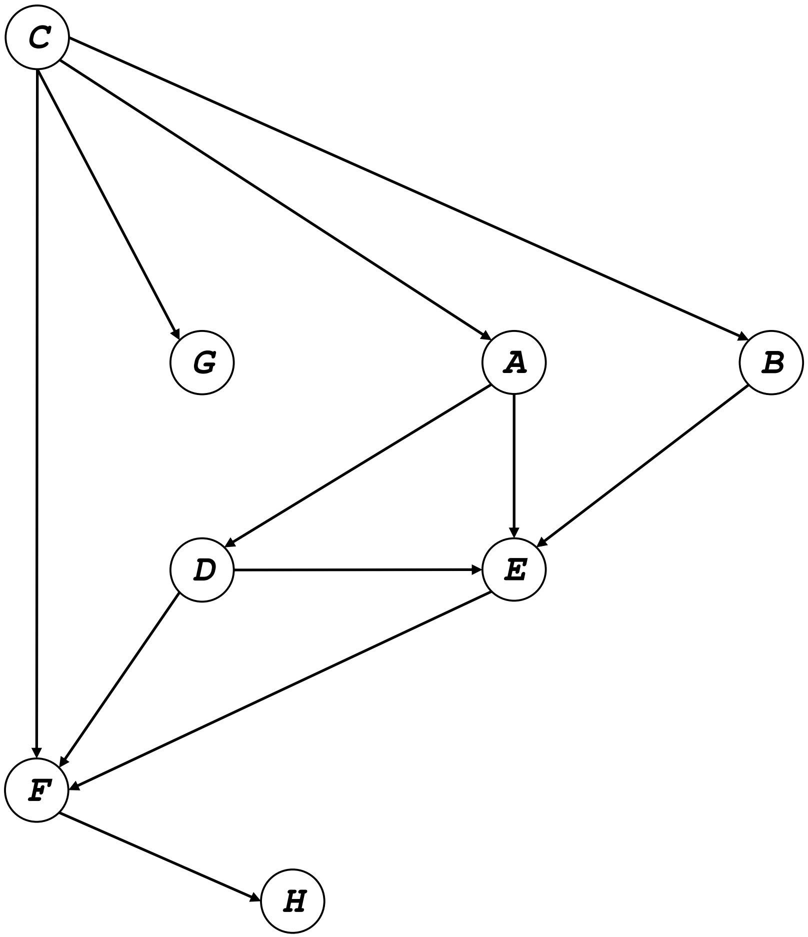
1	0
2	0
3	0
4	0
5	0
6	0

Implementation

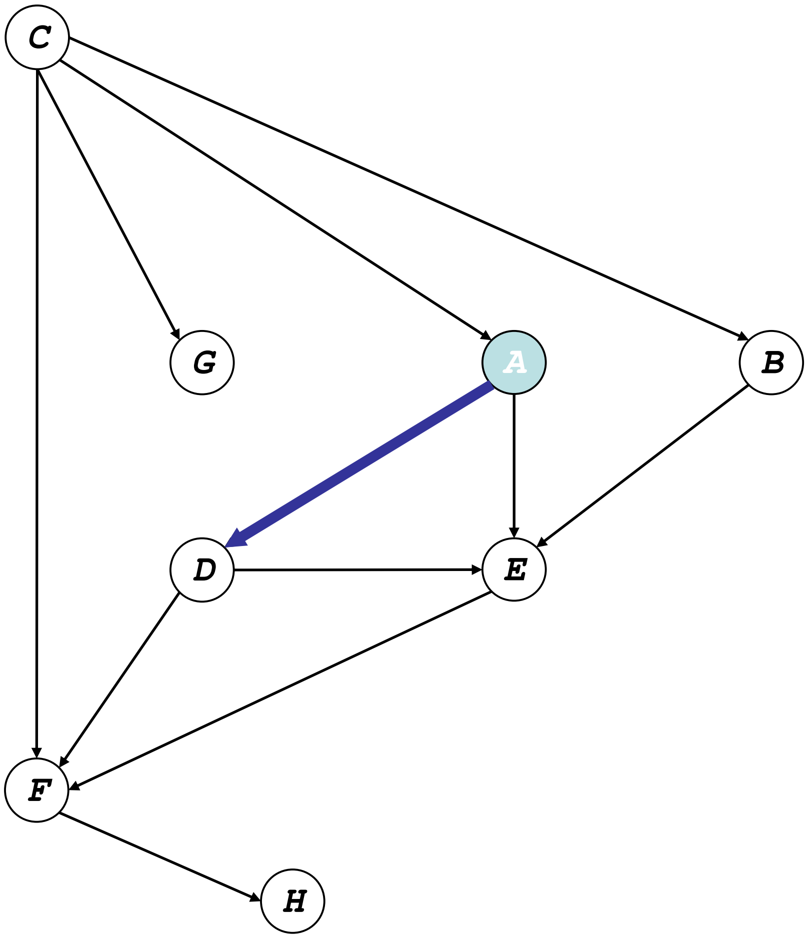
- The queue is now empty, so a topological sort is 1, 6, 2, 5, 3, 4



Another Implementation: DFS

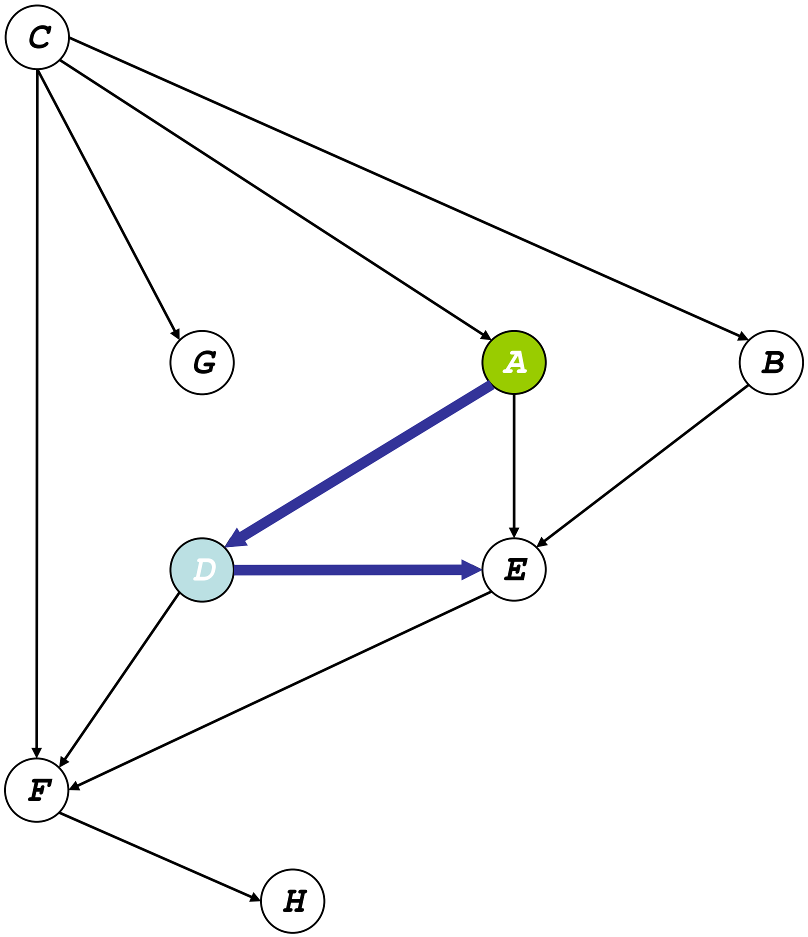


Another Implementation: DFS



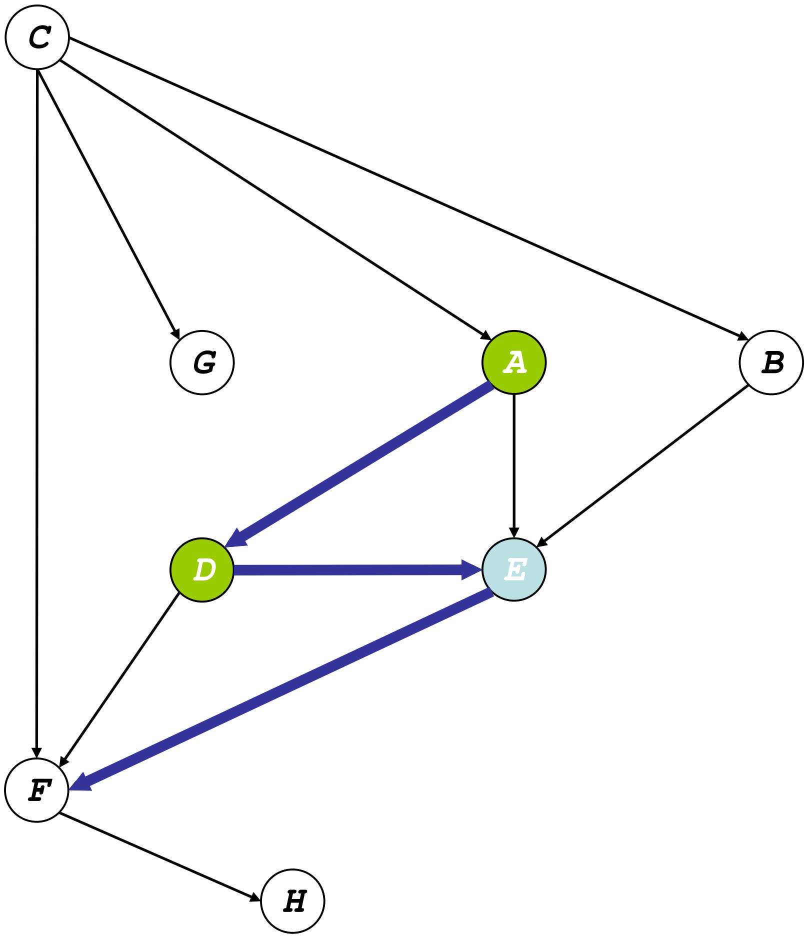
`dfs(A)`

Another Implementation: DFS



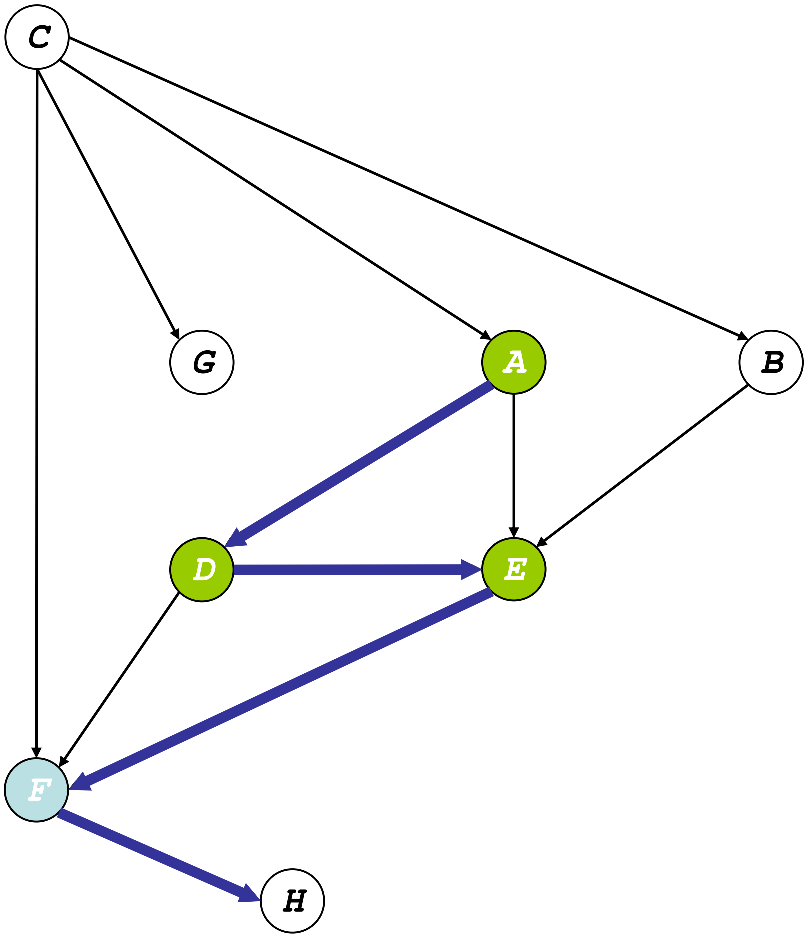

```
dfs(A)
   dfs(D)
```

Another Implementation: DFS



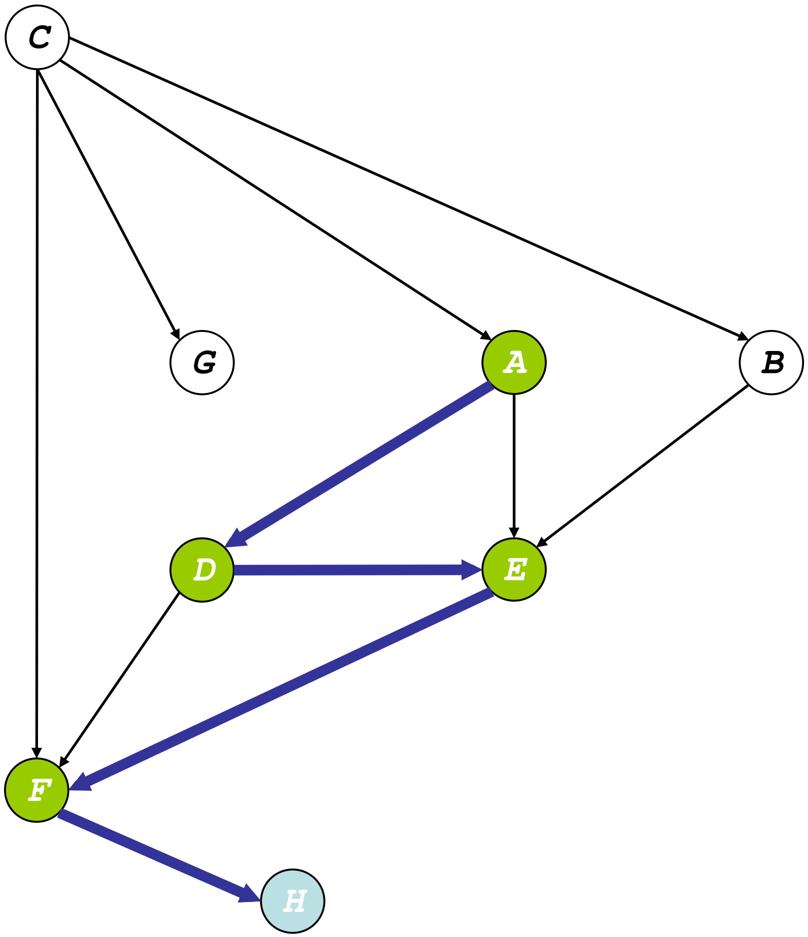

```
dfs(A)
   dfs(D)
      dfs(E)
```

Another Implementation: DFS



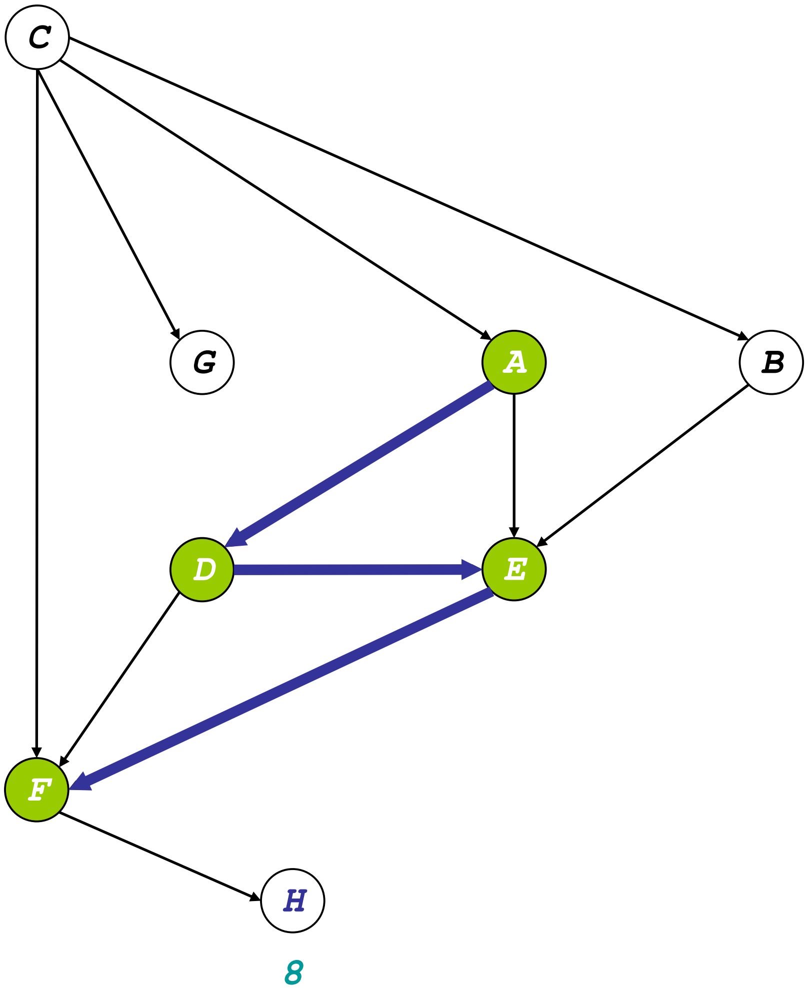

```
dfs(A)
   dfs(D)
      dfs(E)
         dfs(F)
```

Another Implementation: DFS



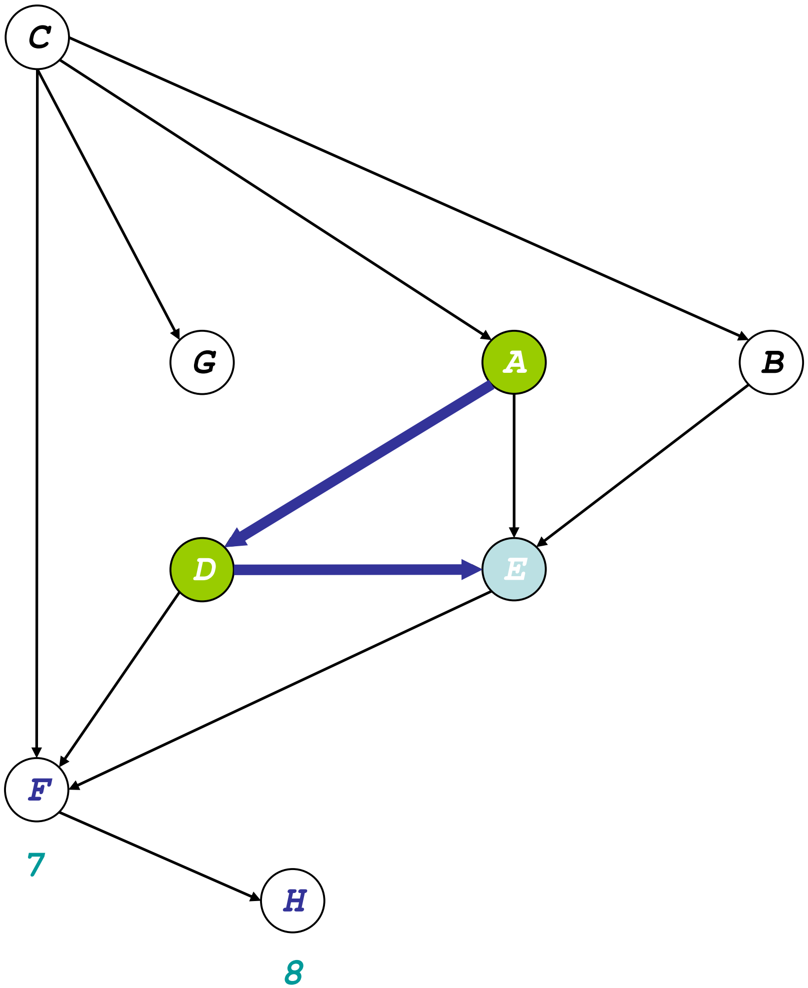

```
dfs(A)
   dfs(D)
      dfs(E)
         dfs(F)
            dfs(H)
```

Another Implementation: DFS



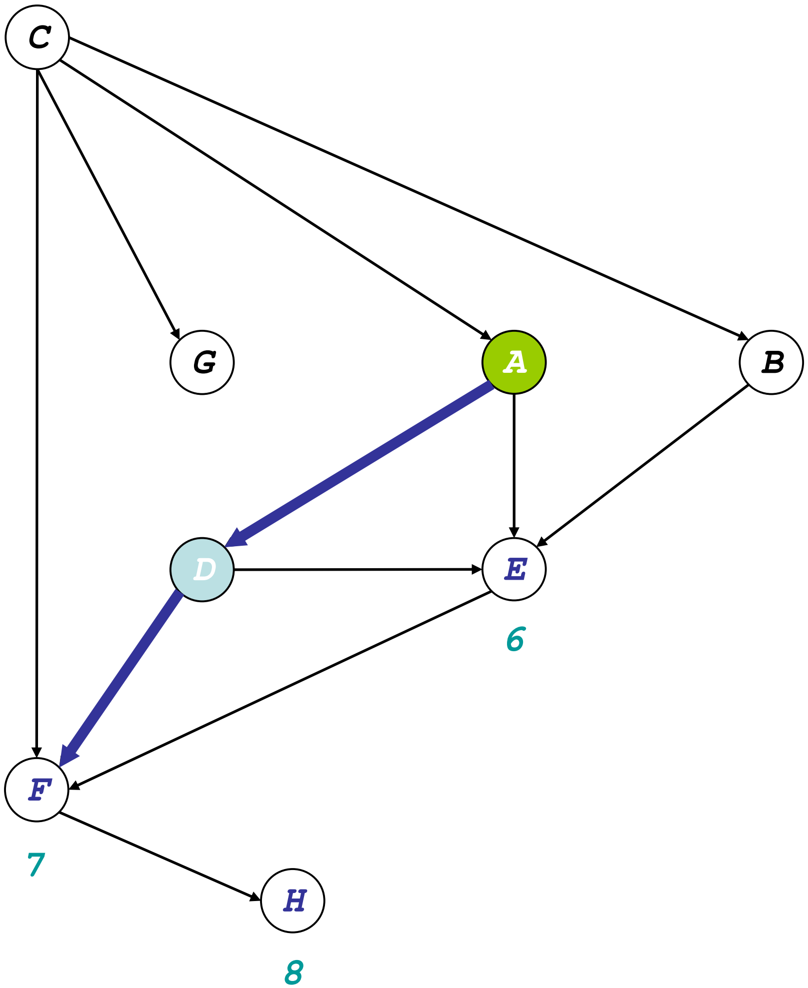

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
```

Another Implementation: DFS



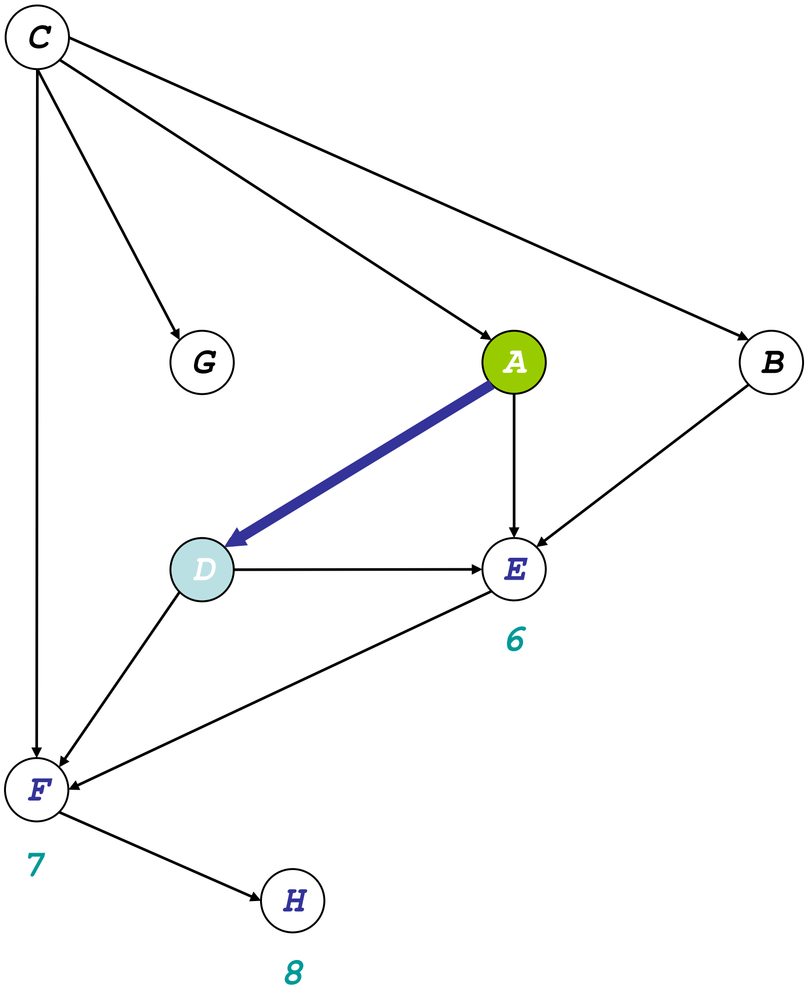

```
dfs(A)
   dfs(D)
      dfs(E)
```

Another Implementation: DFS



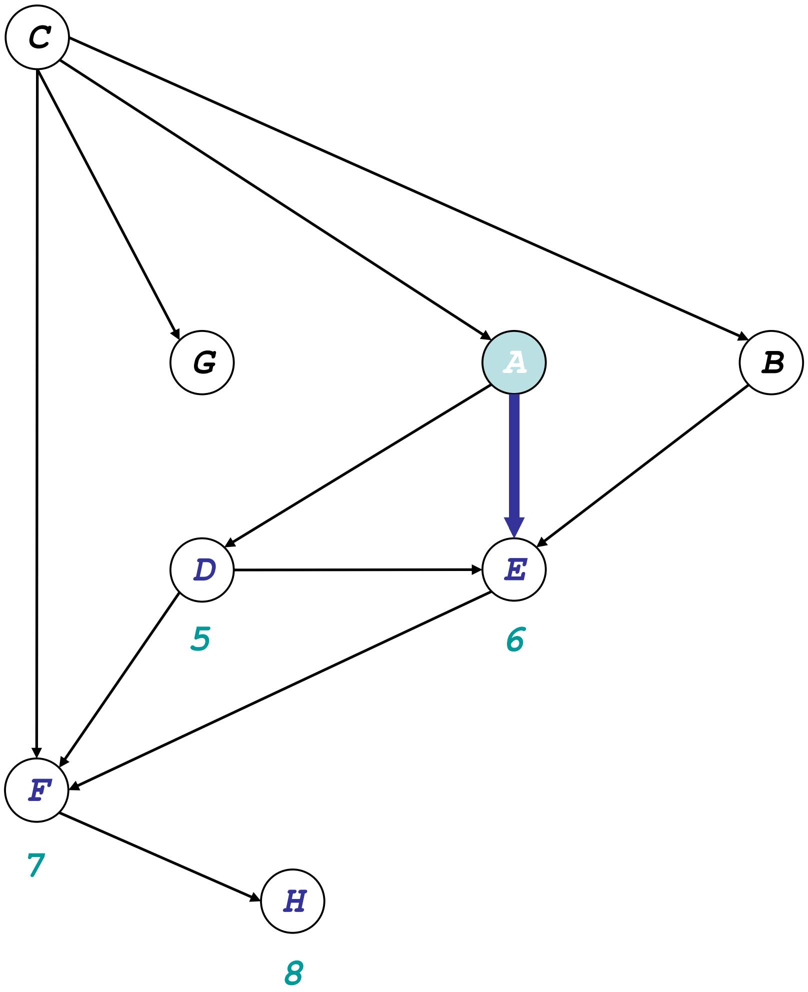

```
dfs(A)
  dfs(D)
```

Another Implementation: DFS



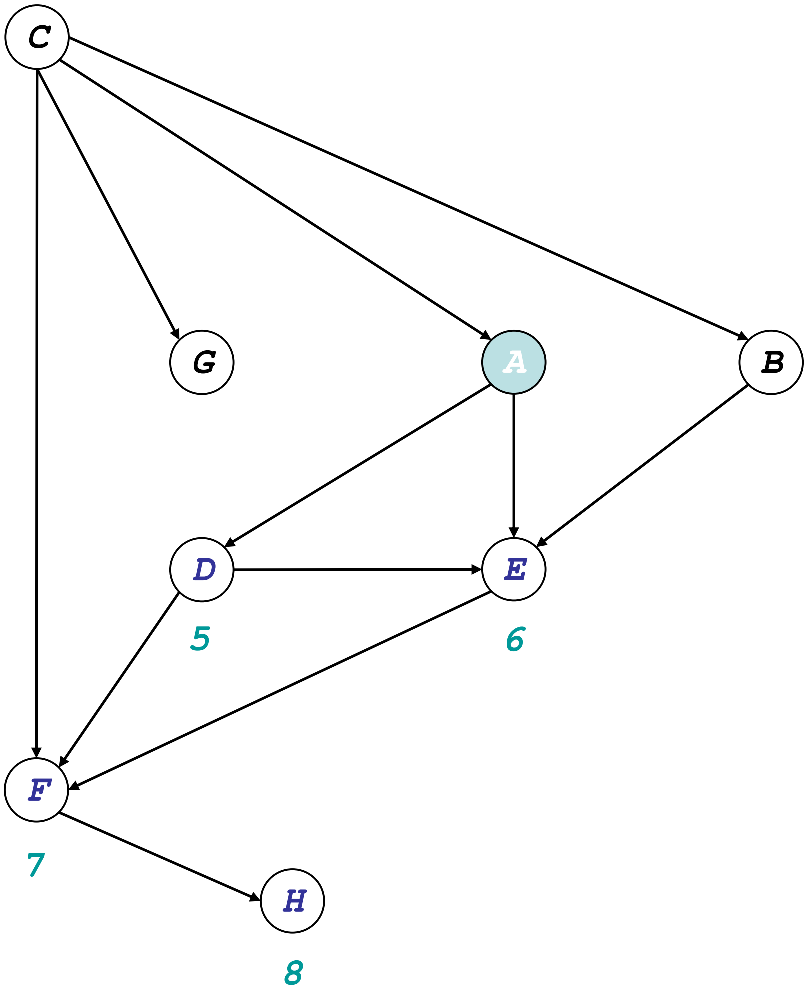

```
dfs(A)
   dfs(D)
```

Another Implementation: DFS



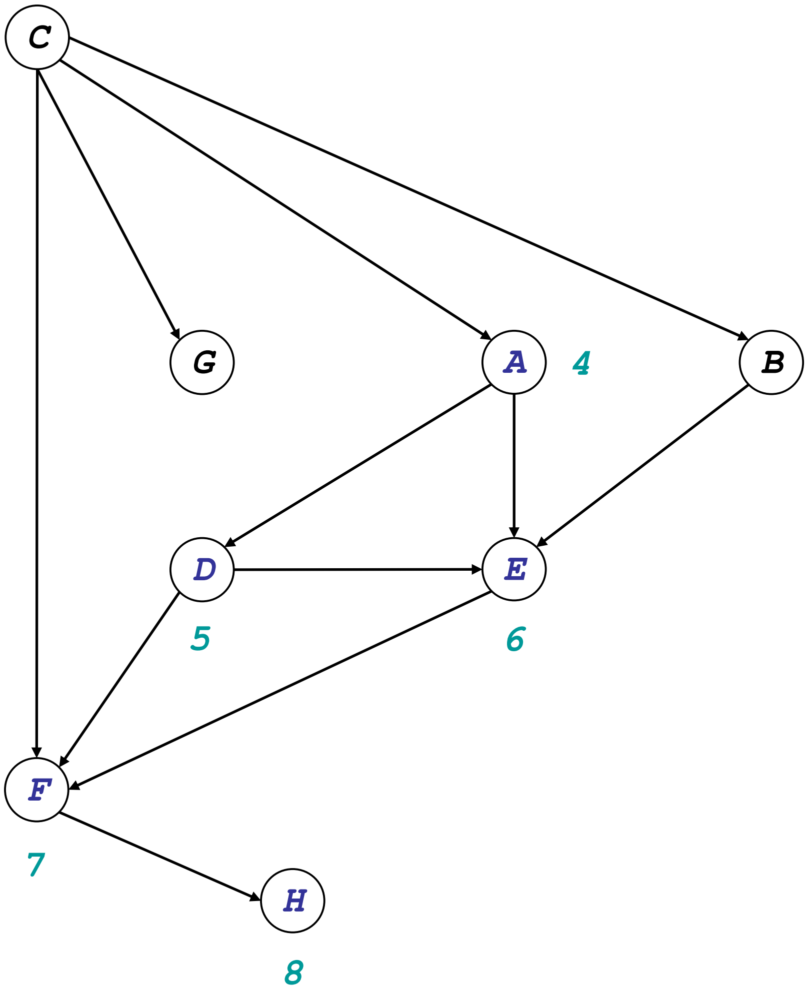
`dfs(A)`

Another Implementation: DFS

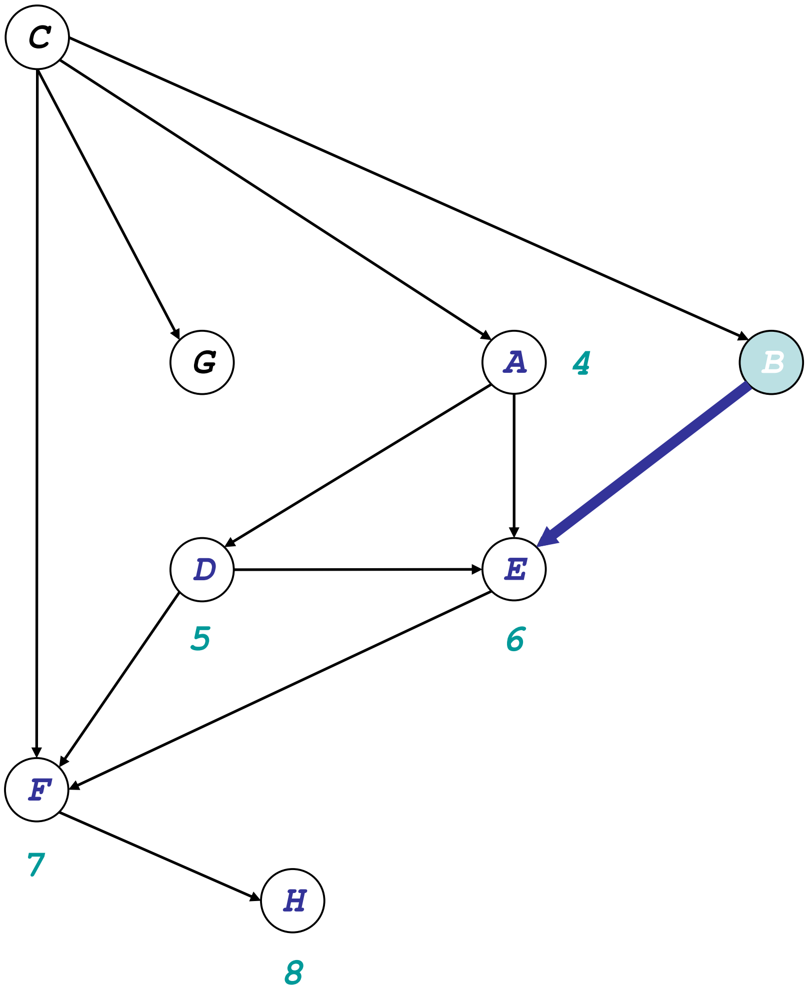


dfs(A)

Another Implementation: DFS

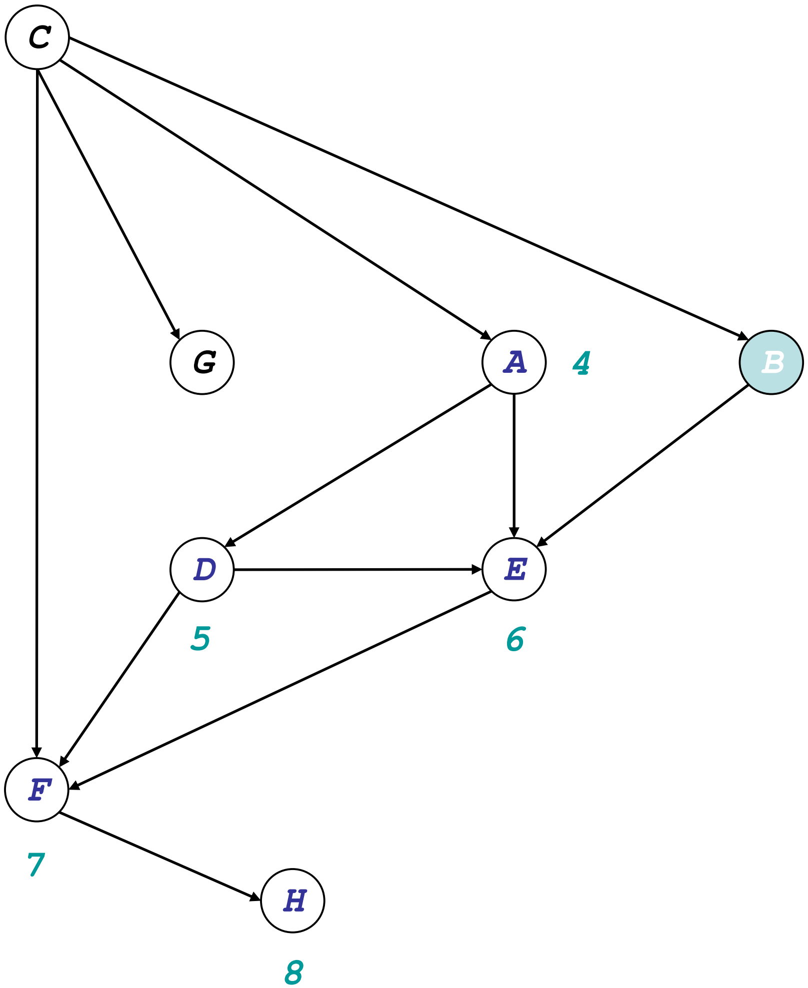


Another Implementation: DFS



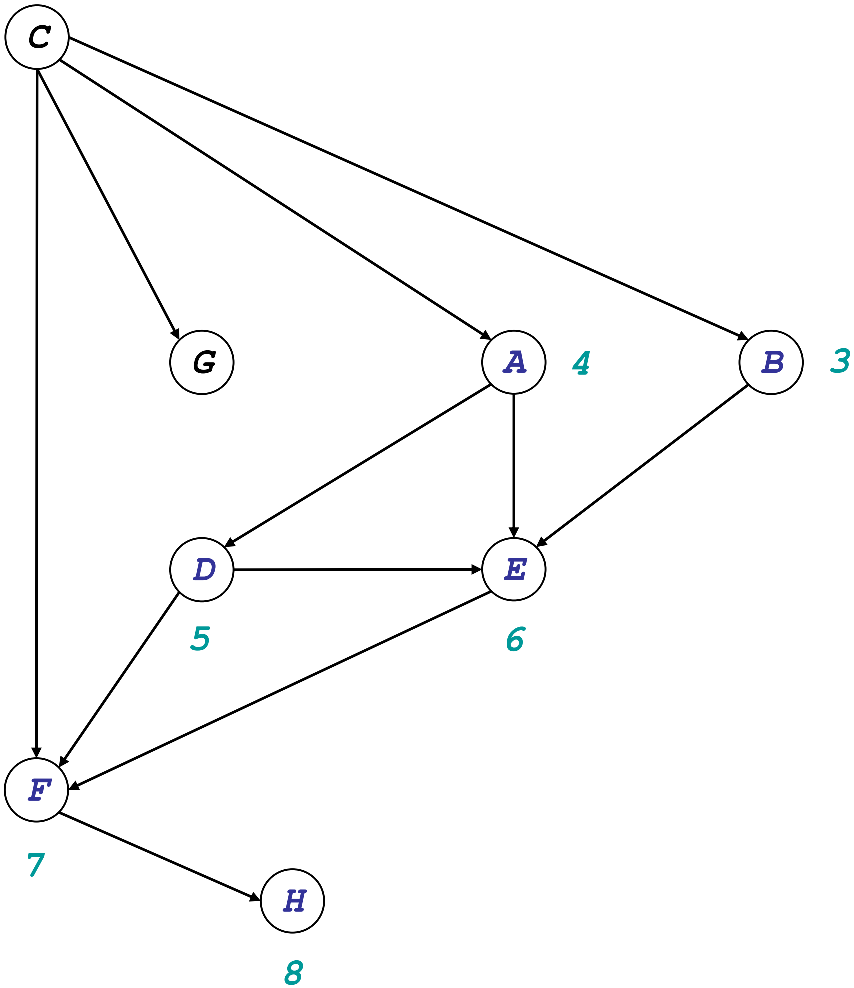
`dfs(B)`

Another Implementation: DFS

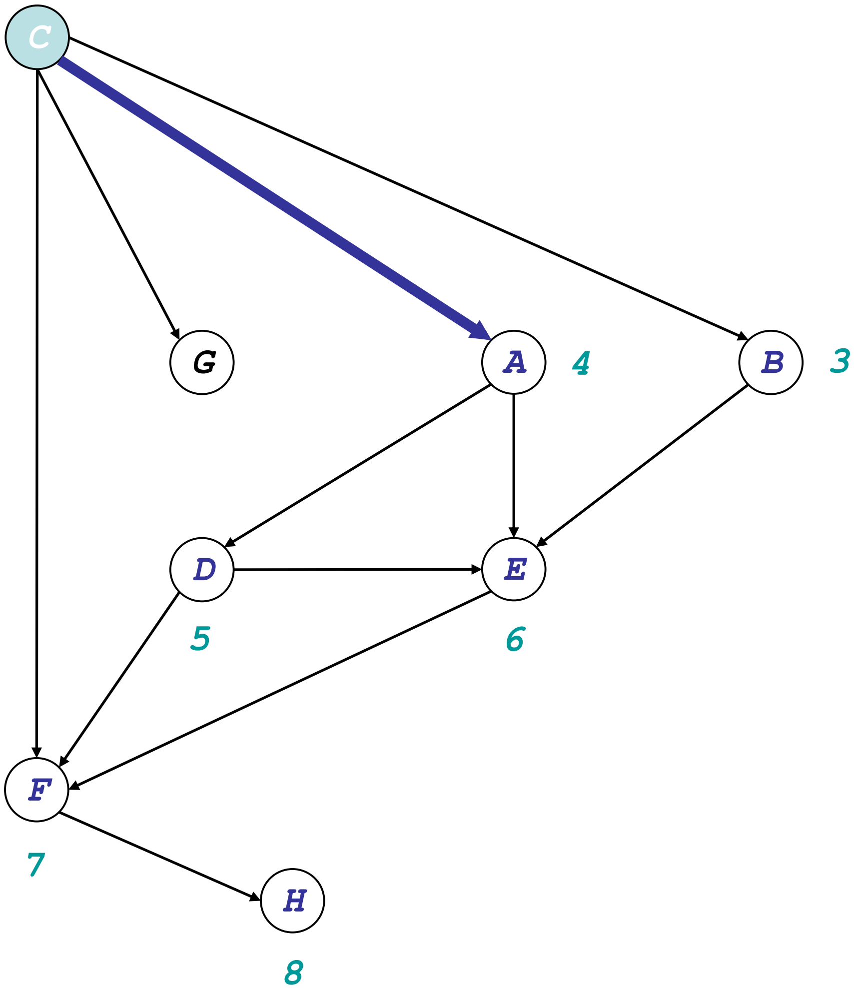


`dfs(B)`

Another Implementation: DFS

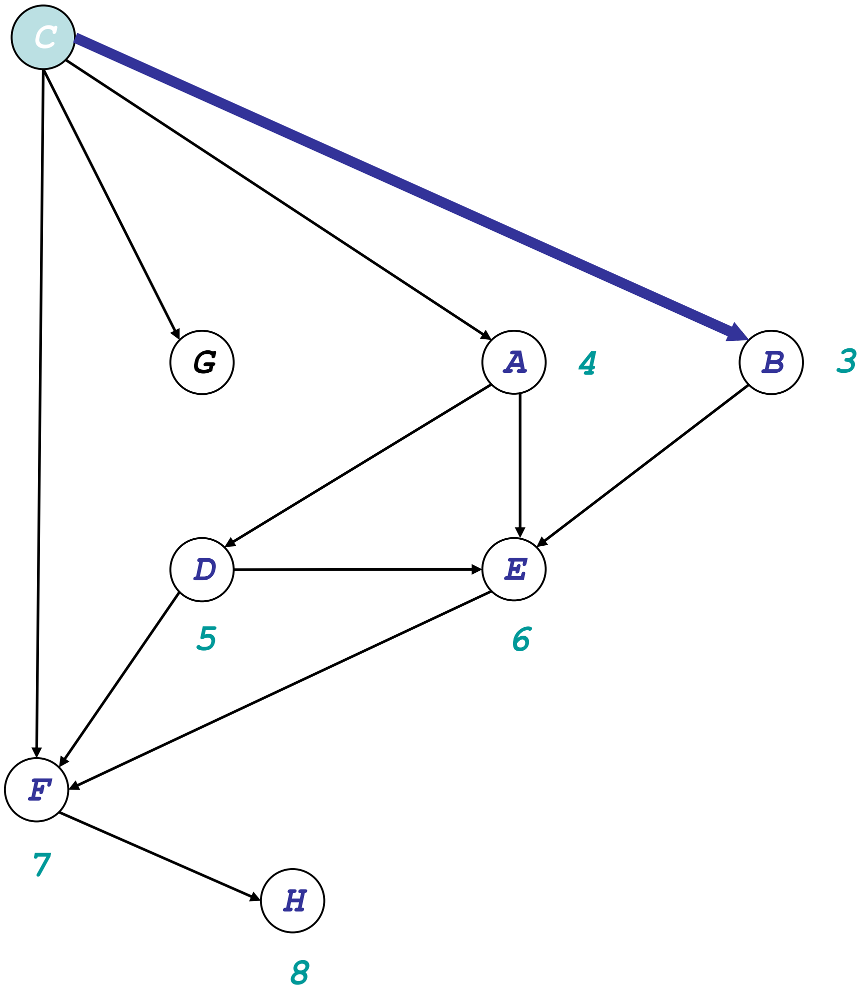


Another Implementation: DFS



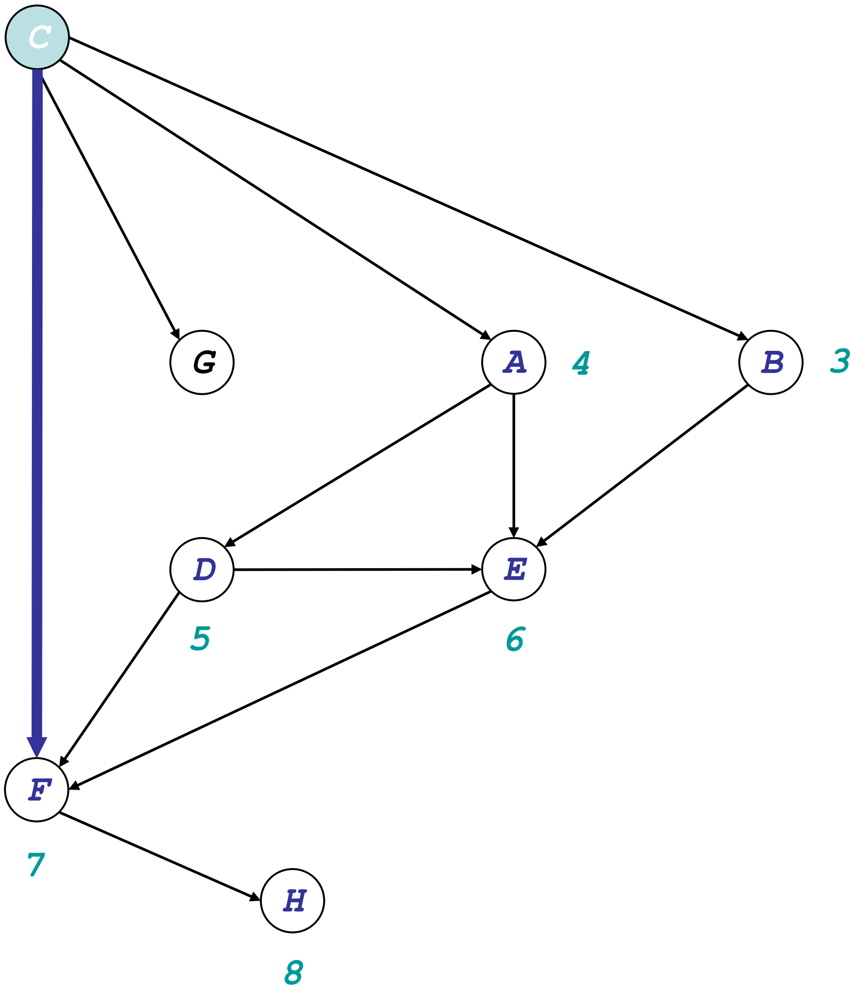
`dfs(C)`

Another Implementation: DFS



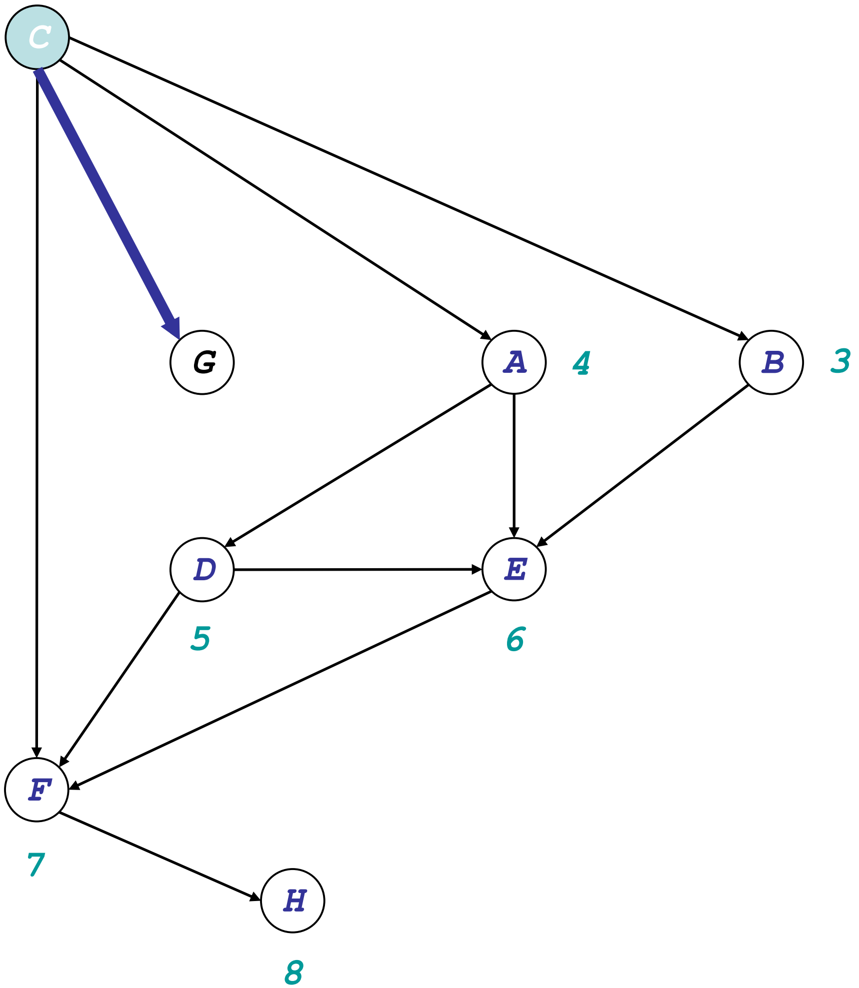
`dfs(C)`

Another Implementation: DFS



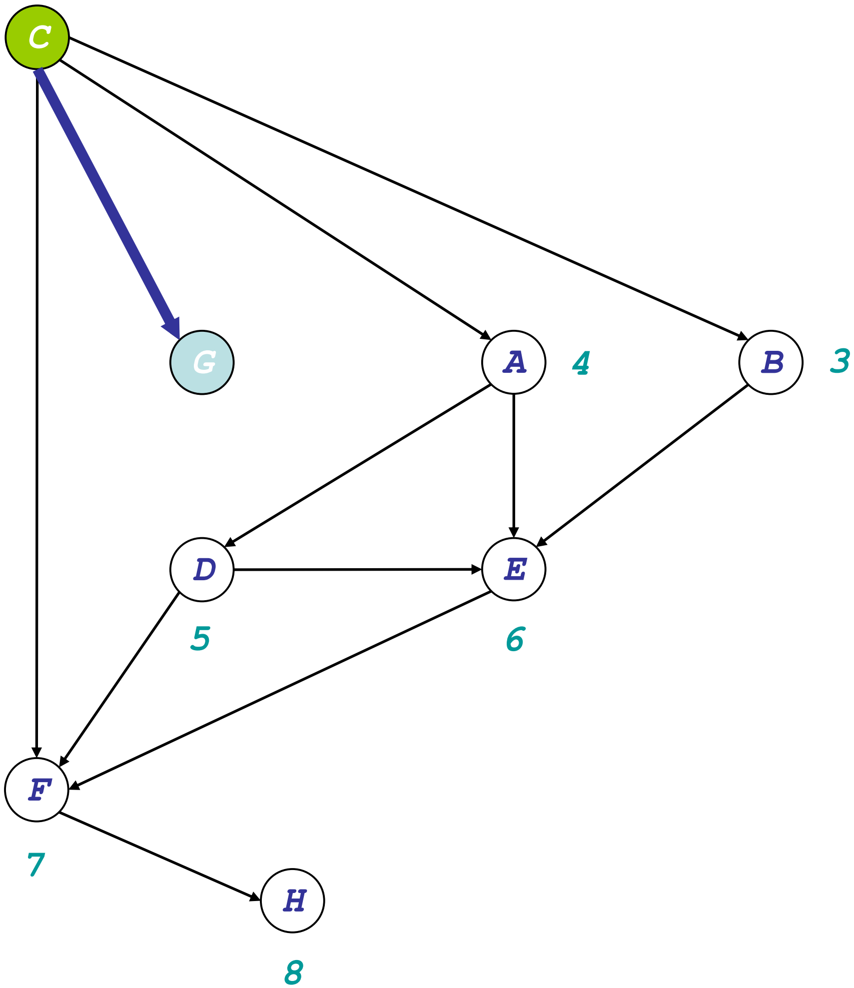
`dfs(C)`

Another Implementation: DFS



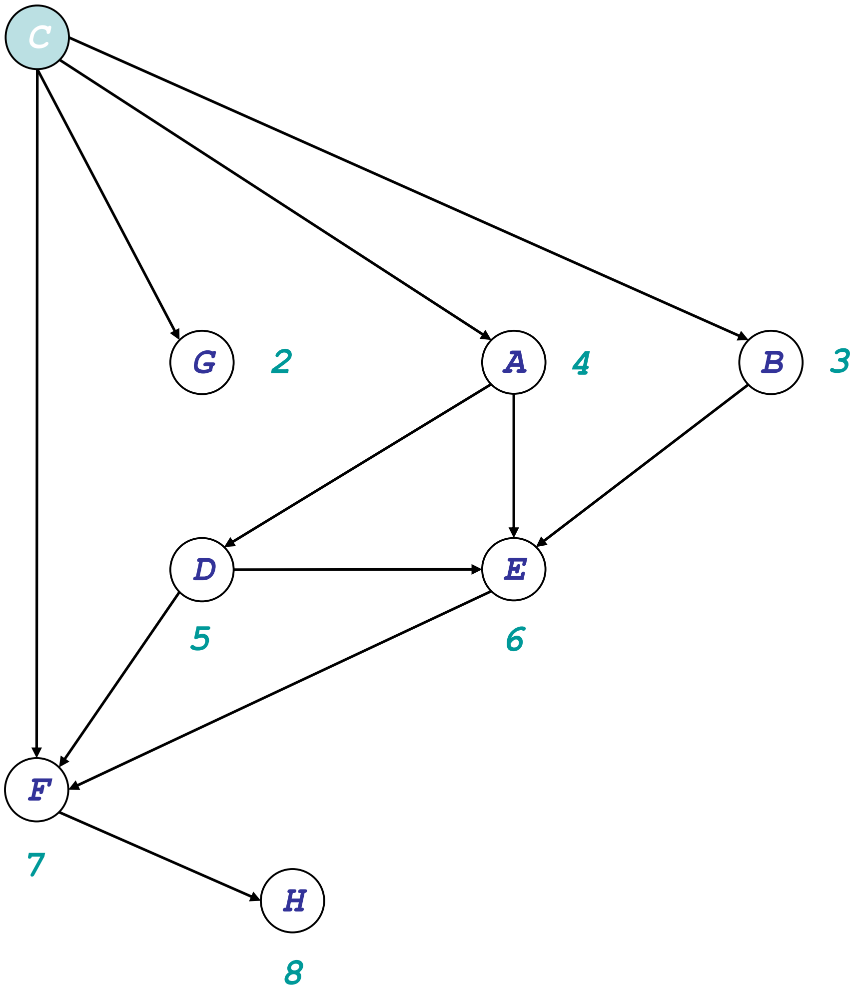
`dfs(C)`

Another Implementation: DFS



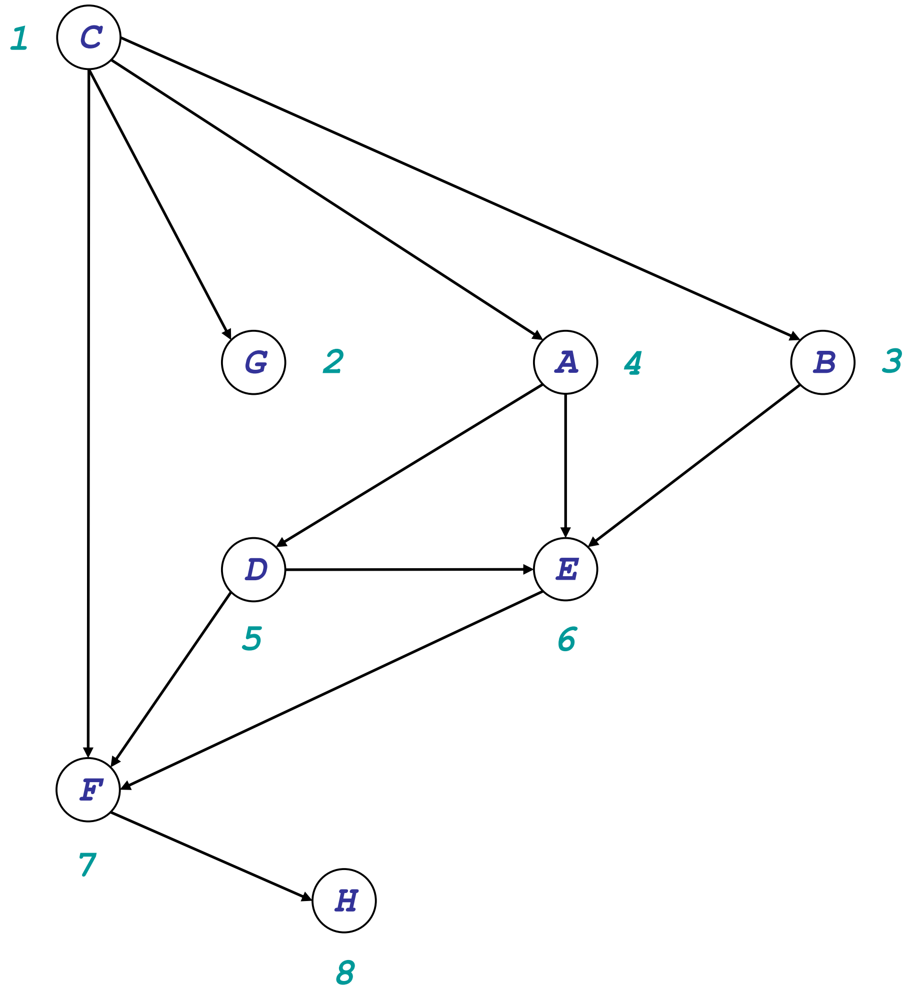

```
dfs(C)
   dfs(G)
```

Another Implementation: DFS



`dfs(C)`

Another Implementation: DFS

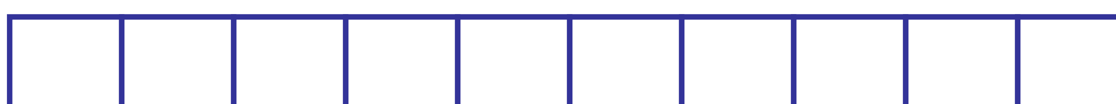


Topological order: C G B A D E F H

Topological Sorting Using DFS

```
Topological_sort(adj)  {
    n = adj.last
    ts = new array(n)
    ts.ind = n  // the index in ts where the next vertex is to be stored in  topological sort
    for i = 1 to n
        visit[i] = false
    for i = 1 to n
        if visit[i]!=true
            dfs_recurs(adj,visit,i,ts)
    return ts
}
dfs_recurs(adj,visit,v,ts) {
    visit[v] = true
    u = adj[v]
    while (u!= null) {
        if (!visit[u.data])
            dfs_recurs(adj,visit,u.data,ts)
        u = u.next
    }
    ts[ts.ind] = v
    ts.ind = ts.ind - 1
}
```

Topological sort array: ts



An Application of Topological Sorting

- **Time table scheduling**
  - **need to schedule list of courses in the order that they could be taken to satisfy prerequisite requirements.**

Course	*Prerequisites*
COMPSCI 100	MATH 120
COMPSCI 150	MATH 140
COMPSCI 200	COMPSCI 100, COMPSCI 150, ENG 110
COMPSCI 240	COMPSCI 200, PHYS 130
ENG 110	*None*
MATH 120	*None*
MATH 130	MATH 120
MATH 140	MATH 130
MATH 200	MATH 140, PHYS 130
PHYS 130	*None*

1 : MATH130

2: MATH 140

3: MATH 200

4: MATH 120

5: COMPSCI 150

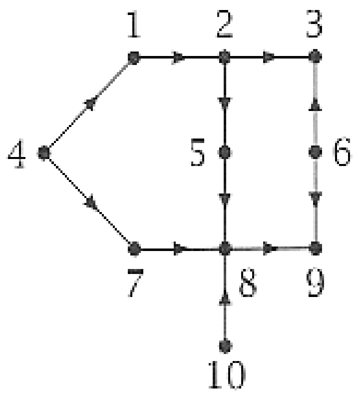
6: PHYS 130

7: COMPSCI 100

8: COMPSCI 200

9: COMPSCI 240

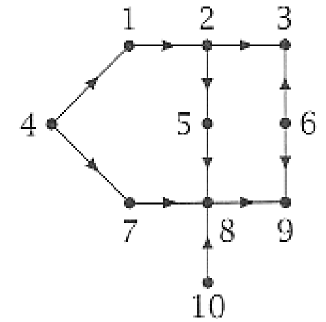
10: ENG 110



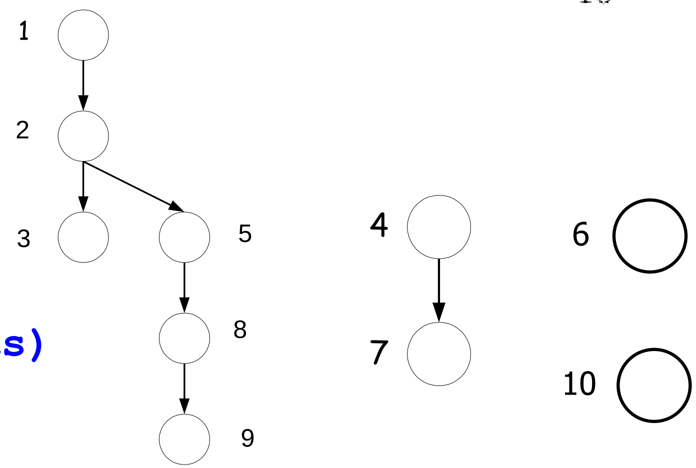
An Application of Topological Sorting

```
Topological_sort(adj)  {
    n = adj.last
    ts = new array(n)
    ts.ind = n  // the index in ts where the next vertex is to be stored in  topological sort
    for i = 1 to n
        visit[i] = false
    for i = 1 to n
        if visit[i]!=true
            dfs_recurs(adj,visit,i,ts)
    return ts
}
dfs_recurs(adj,visit,v,ts) {
    visit[v] = true
    u = adj[v]
    while (u!= null) {
        if (!visit[u.data])
            dfs_recurs(adj,visit,u.data,ts)
        u = u.next
    }
    ts[ts.ind] = v
    ts.ind = ts.ind - 1
}
```

Input Graph



dfs trees



Topological sort array: ts

10	6	4	7	1	2	5	8	9	3

Learning Takeaway

- **Topological sorting can be implemented using**
  - **queue**
  - **DFS: recall the template method**

- **Applications include scheduling of tasks and time tables.**

Algorithm Design and Applications

Types of Algorithms

- What type of algorithm to use depends very much on the application.
- Some choices may result in lower running time complexity and/or memory storage complexity.
- Need to choose carefully.

- We study 3 different algorithm designs:
  - Greedy algorithms
  - Backtracking algorithms
  - Dynamic programming

Greedy Algorithms

The Greedy Technique

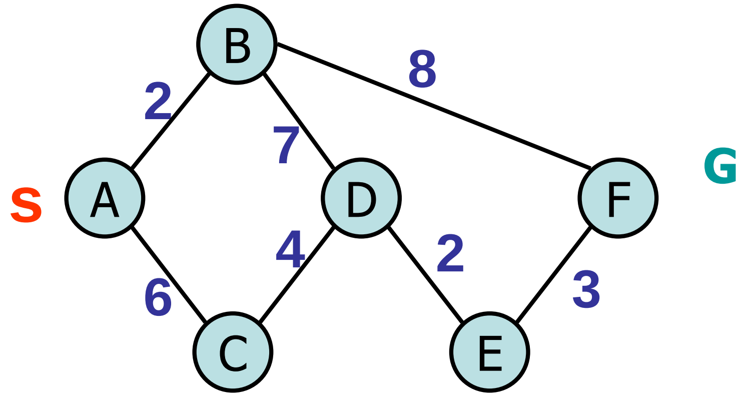
- A greedy algorithm
  - ☞ Builds a solution to a problem in steps
  - ☞ In each step, it adds a part of the solution
  - ☞ The part of the solution to be added is determined by a greedy rule
    - ✓ Greedy rule: if given a choice, it operates by choosing locally most valuable alternative.
- A greedy algorithm may or may not be optimal (best possible)

Applications of the Greedy Technique

- **Shortest Path**
  - **Dijkstra's Algorithm**

- **Minimum Spanning Trees**
  - **Kruskal's Algorithm**
  - **Prim's Algorithm**

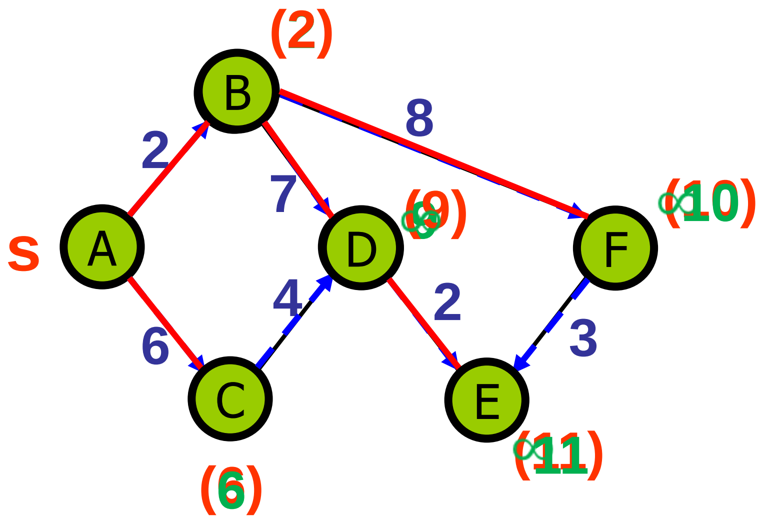
Shortest Path Problem



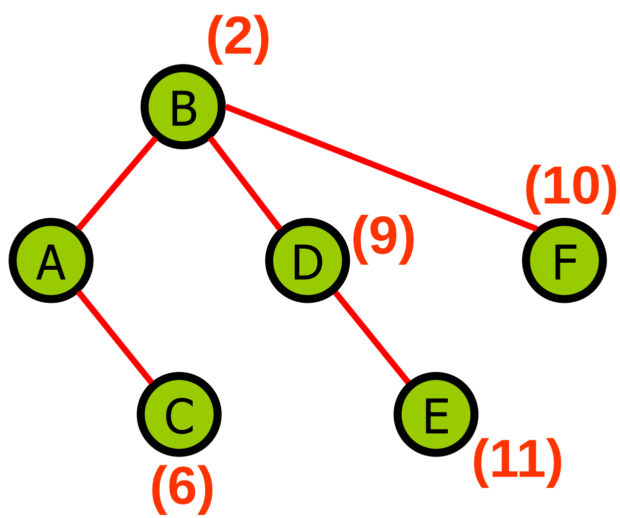
- **Shortest path**
  - **Path of minimum distance between a given pair of vertices**
- **Single-Source Shortest Path Problem**
  - **Find shortest paths from a given vertex s to <u>all</u> the other vertices**

Dijkstra's Algorithm

G



SPT



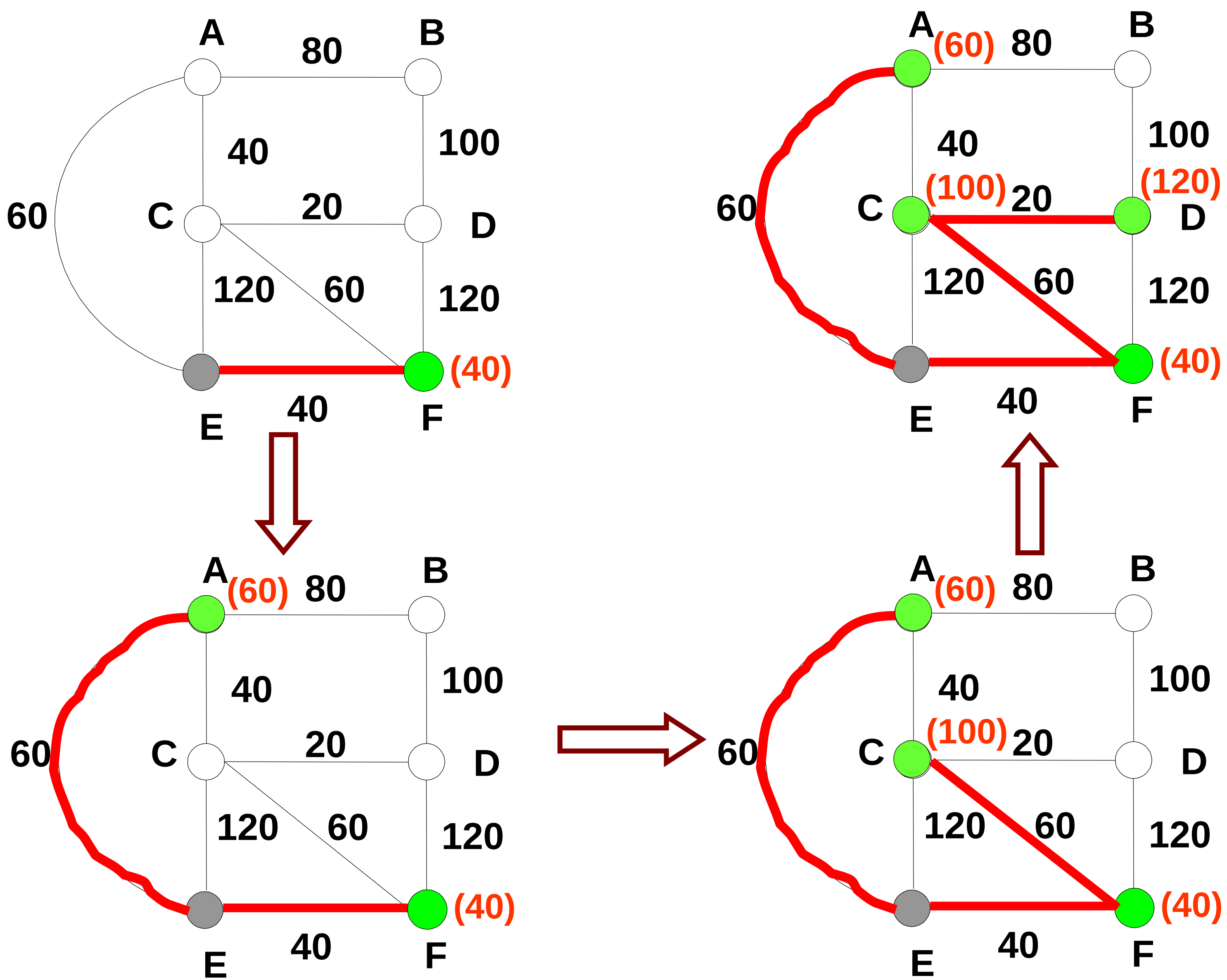
Input: weighted graph with non-negative weights

Dijkstra's Algorithm (s)

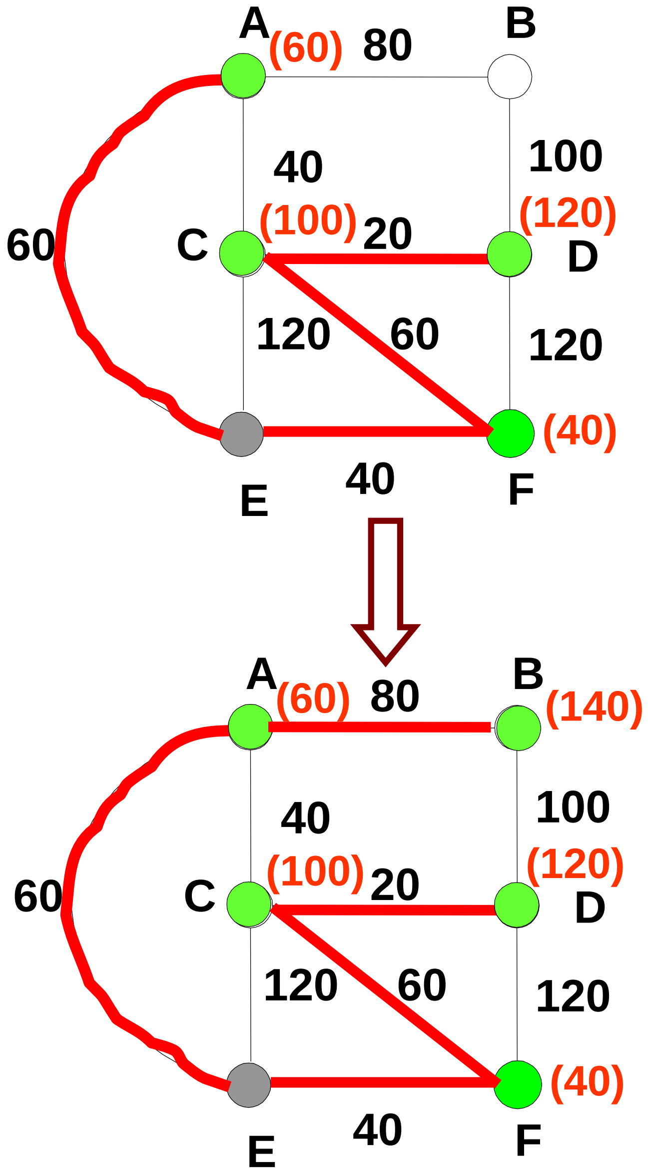
1) **Add a minimum edge starting from s to an empty SPT**
2) **If the number of the edges of SPT is less than n-1, keep growing tree SPT by repeatedly adding edges which can extend the paths from s in SPT as short as possible.**

SPT always remains as a tree when Dijkstra's Algorithm runs

Example


A
B
80
40
100
60
C
20
D
120
60
120
(40)
E
40
F
A
(60)
80
B
40
100
60
C
20
D
120
60
120
(40)
E
40
F
A
(60)
80
B
40
100
60
C
(100)
20
D
120
60
120
(40)
E
40
F
A
(60)
80
B
40
100
(120)
60
C
(100)
20
D
120
60
120
(40)
E
40
F

Example



Shortest Path	Length
E	0
E,F	40
E,A	60
E,F,C	100
E,F,C,D	120
E,A,B	140

Time Complexity of Dijkstra's Algorithm

Input: G = (V,E) a weighted graph, with all edge weights non-negative. If there is no edge between u and v, take the weight w(u,v) = ∞

```
V_T  = {1};  k = 0;  s = 1;
for j = 1 to n
      dist[j] = w(1,j)
while k < n  do{
      min = ∞
      for ( each j ∈ V − V_T ) {
           if (dist[j]  < min) {
                 min = dist[j]
                 new = j
           }
      }
      V_T = V_T ∪ {new}
      k = k + 1
      for ( each j ∈ V − V_T ) {
           if (dist[new] + w(new, j)  <  dist[j]) {
                 dist[j]  = dist[new] + w(new, j)
           }
      }
}
```

O(n)

n iterations

O(n)

This is known as relaxing the edge (new, j). dist[j] is an estimate of the true shortest path weight $\delta(s, j)$ from s to j.

O(n)

Overall complexity = O(n) + n*(O(n)+O(n)) = O(n^2)

Inductive Proof of Dijkstra's Algorithm (Optional)

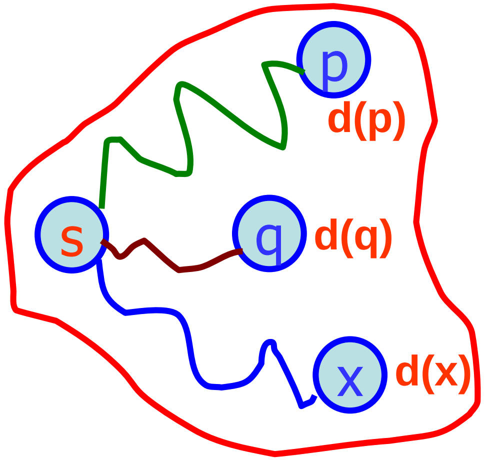
Show that at each step, algorithm maintains a SPT to all added vertices.
Condition: all the weights are non-negative.

Basis Step:

If n=1, the tree contains zero edge

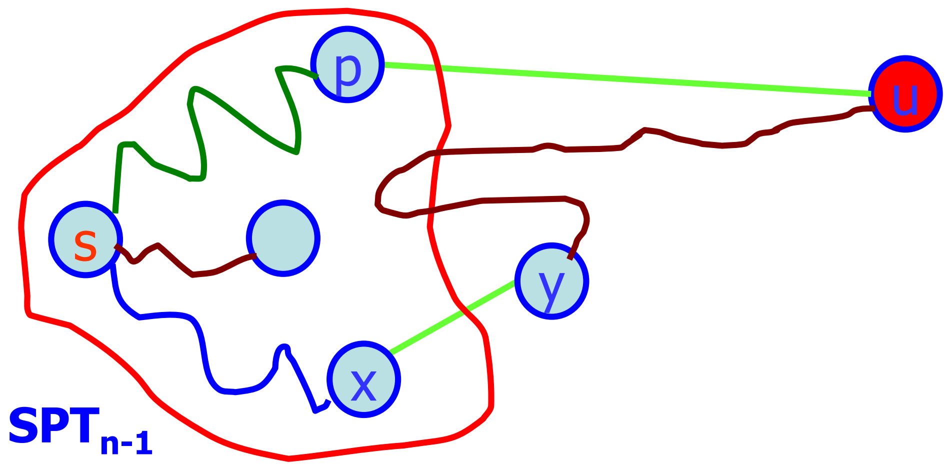
Inductive Step:

Assume Dijkstra's algorithm obtains the right shortest paths for the first n-1 vertices.



Inductive Proof of Dijkstra's Algorithm (Optional)

For the next vertex u added by the algorithm:



Suppose that $s \rightsquigarrow p \rightarrow u$ is not a shortest path. Then, there exists a shortest path $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$, where $x \neq p$ is in SPT$_{n-1}$ and y is outside (possibly $u = y$).

Claim 1: $p = s \rightsquigarrow x \rightarrow y$ is a shortest path
Proof: If there is another path p' from s to y such that $w(p') < w(p)$, then the path $(p',\ y \rightsquigarrow u)$ has weight strictly less than $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$, a contradiction.

Inductive Proof of Dijkstra's Algorithm (Optional)

Induction hypothesis: $\mathbf{dist}[x] = \delta(s, x)$

Claim 2: dist[y] = $\delta(s, y)$ at all times after relaxing the edge (x, y).

Proof: From relaxing the edge, we have dist[y] $\leq \mathbf{dist}[x] + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$ from Claim 1. But $\delta(s, y) \leq \mathbf{dist}[y]$ by definition, so the claim holds.

Therefore, we have $\mathbf{dist}[y] = \delta(s, y) \leq \delta(s, u)$ because weights are non-negative.

In Dijkstra's algorithm, since u is chosen as the next vertex to add, we have $\mathbf{dist}[u] \leq \mathbf{dist}[y] \leq \delta(s, u)$, and since $\delta(s, u) \leq \mathbf{dist}[u]$ by definition, we have $\mathbf{dist}[u] = \delta(s, u)$. This is a contradiction to the assumption that $s \rightsquigarrow p \rightarrow u$ is not a shortest path.

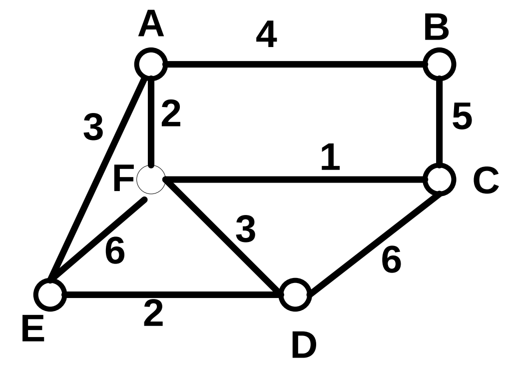
This shows that each step of the Dijkstra's algorithm finds a correct shortest path.

BFS vs Dijkstra

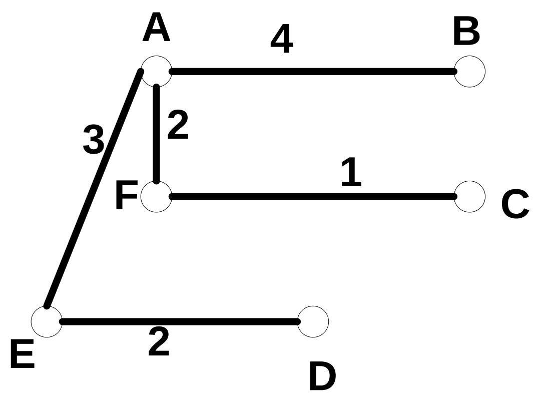
- **Should we use BFS or Dijkstra's Algo to find shortest paths ??**
  - **Unweighted graph**
    - **Use BFS**
      - **Complexity O(n+m)**
  - **Weighted graph with non-negative weights**
    - **Use Dijkstra's Algo**
      - **Complexity $O(n^2)$**

Minimum Spanning Trees

- A graph is called a **tree** if it is *connected* and it contains no cycle.
- A **spanning tree** of a graph G is a subgraph of G, which is a tree and contains all vertices of G.
- Minimum Spanning Tree (MST)
  - Spanning tree of a weighted graph with *minimum total edge weight*.
  - Different from the shortest path tree!

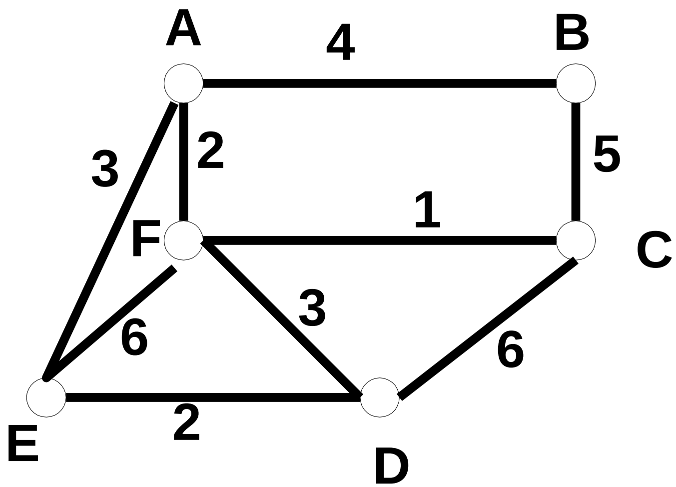


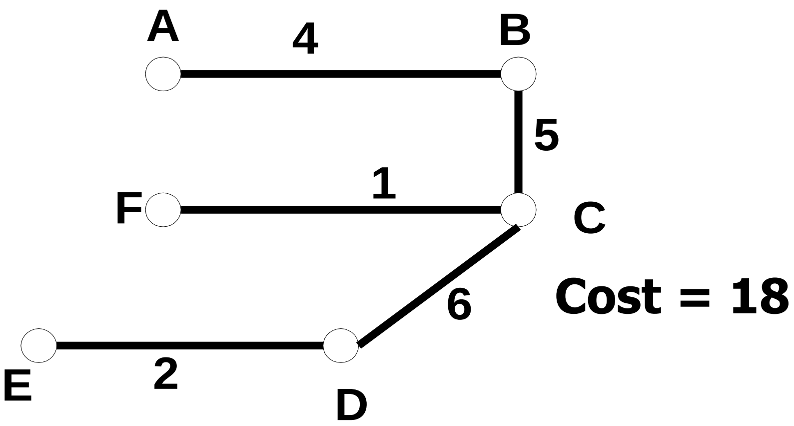
A weighted graph



A minimum spanning tree (weight = 12)

An Application of MST

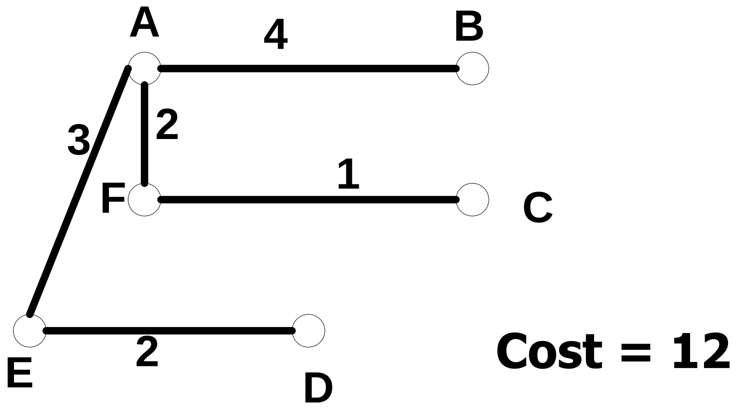




Each node represents a city

Weight of each edge: cost of building a road connecting two cities

Problem: to build enough roads so that each pair of cities will be connected and to use the lowest cost possible



Origin of MST

- **Otakar Borůvka, Czech mathematician developed first algorithm to find MST in 1926.**

- **Trying to solve a very practical problem: what is the most economical electrical power network to cover the country of Moravia (now part of Czech Republic).**

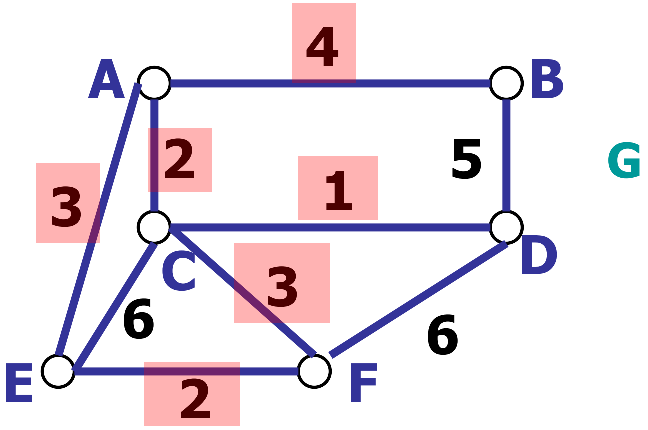
Constructing MST

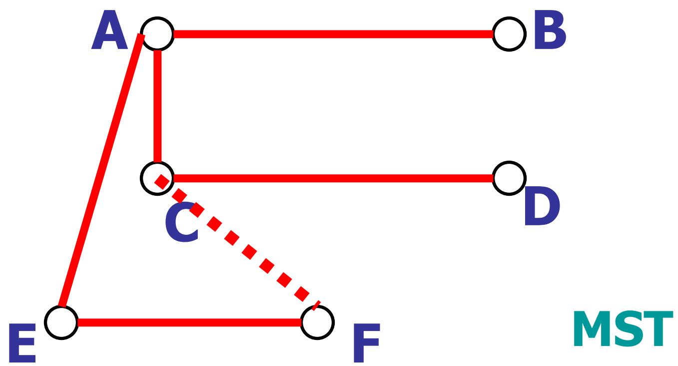
- A Minimum Spanning Tree can be constructed using one of the following two popular algorithms
  - ☞ Kruskal's Algorithm
  - ☞ Prim's Algorithm

Kruskal's Algorithm

Kruskal's Algorithm

1) Add all vertices of G in the MST
2) Add an edge of minimum weight of G to the MST
3) If the number of edges of MST is less than n-1, repeatedly add an edge of next minimum weight of G that does not make a cycle to the MST



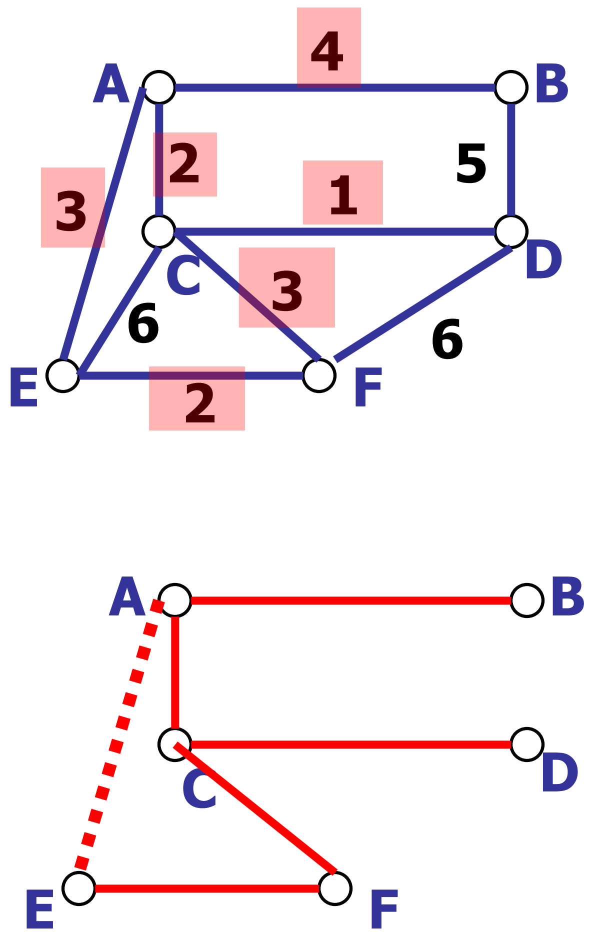


MST need not be a tree until the completion of Kruskal's Algorithm

Kruskal's Algorithm

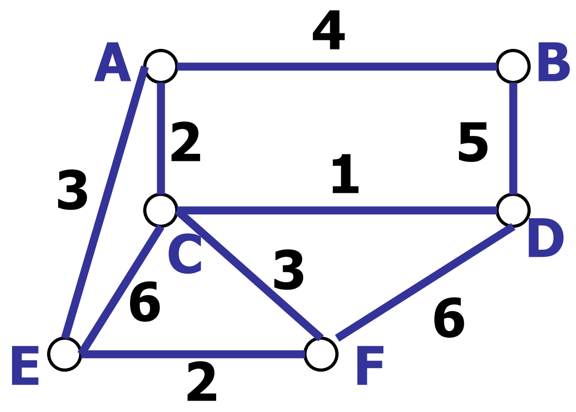
<u>**Kruskal's Algorithm**</u>

1) Add all vertices of G in the MST
2) Add an edge of minimum weight of G to the MST
3) If the number of edges of MST is less than n-1, repeatedly add an edge of next minimum weight of G that does not make a cycle to the MST

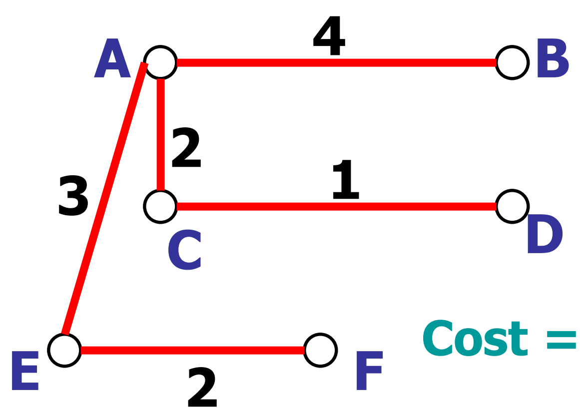


Another MST – same total weight as first one

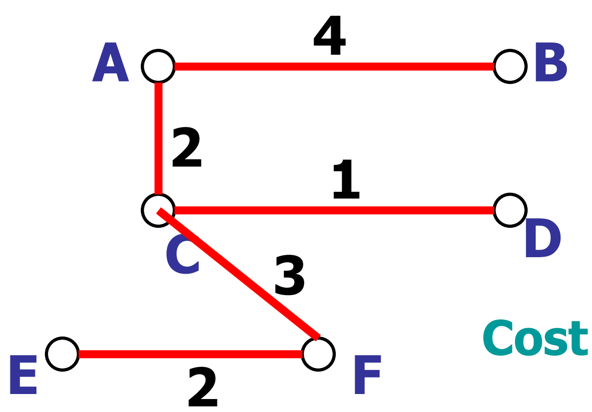
Kruskal's Algorithm



Minimum Spanning Trees are unique?

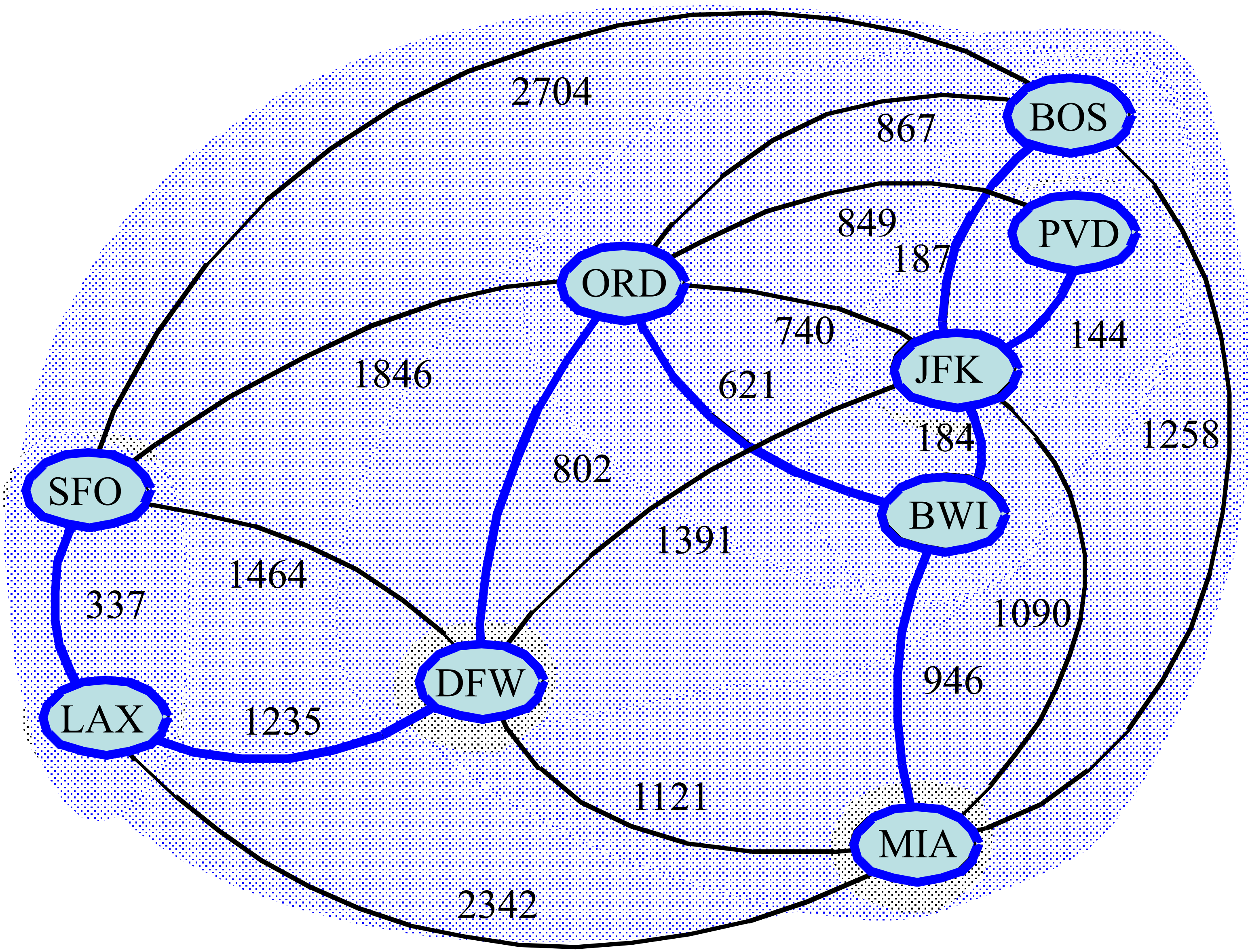


Cost = 12

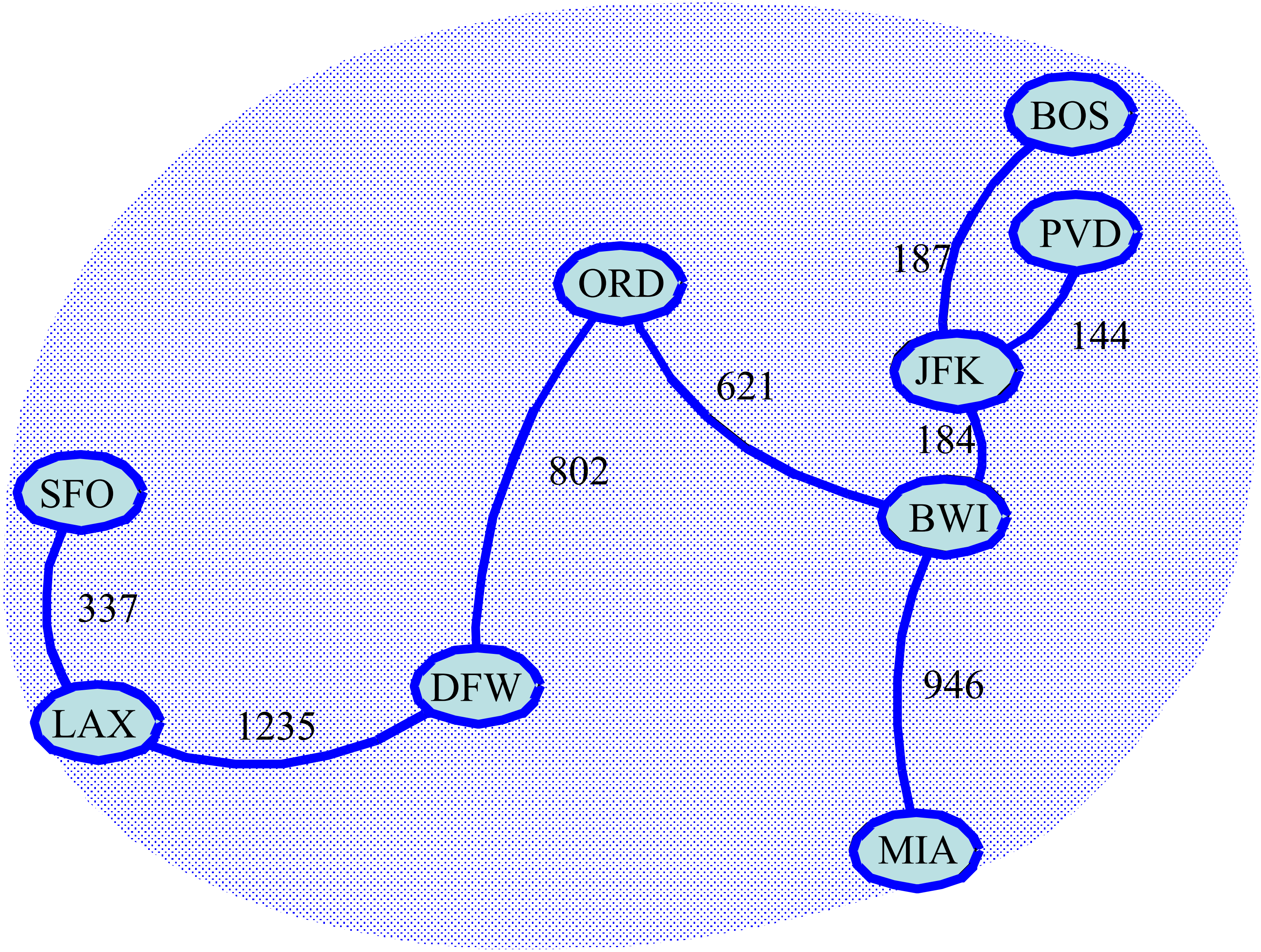


Cost = 12

Example


2704
867
BOS
849
PVD
187
ORD
740
144
1846
JFK
621
184
1258
SFO
802
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

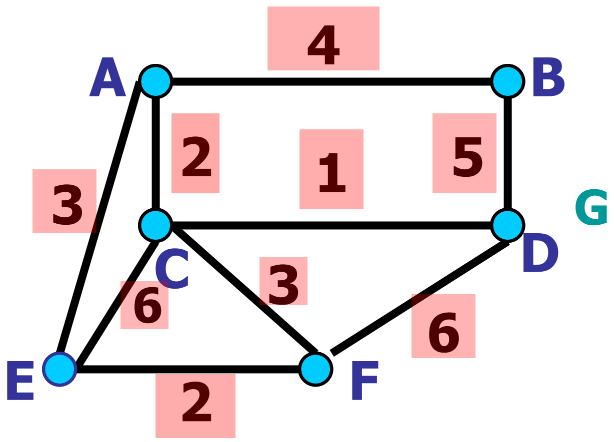
Example

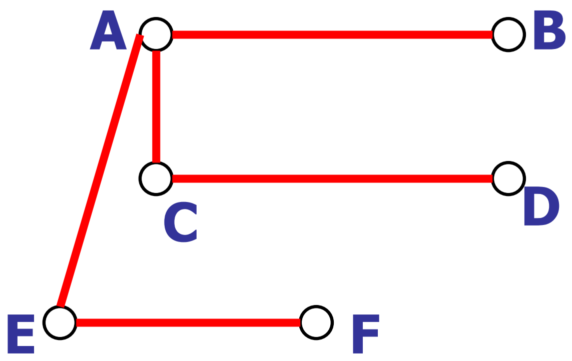

BOS
PVD
187
ORD
144
JFK
621
184
802
SFO
BWI
337
946
DFW
LAX
1235
MIA

Prim's Algorithm

<u>**Prim's Algorithm**</u>

1) Add one vertex of G in the MST
2) If the number of edges of MST is less than n-1, repeatedly add an edge of next minimum weight of G to the MST which has one vertex in MST and another not in MST.





Start vertex = E

MST

MST always remains as a tree when Prim's Algorithm runs

Correctness of Kruskal and Prim (Optional)

- **Both Kruskal and Prim's algorithms build up a set of edges A.**
- **Idea: ensure that at every step, A is a subset of some MST. Why does this hold in Kruskal and Prim's algorithm?**

$G = (V, E)$ is a connected, weighted graph.

Theorem. Suppose that A is a subset of some MST. Let S be a set of vertices of G such that no edge in A has one endpoint in S and the other endpoint in $V - S$ (no edge in A crosses from S to $V - S$). Let (u, v) be a minimum weighted edge crossing from S to $V - S$. Then $A \cup (u, v)$ is a subset of some MST.

Prim's algorithm: Initially, set of edges is empty, so A belongs to a MST. At each subsequent step, set S in Theorem to be the tree constructed by the algorithm. From Theorem, A belongs to some MST at every step. When $n - 1$ edges have been added, the resulting tree is a MST.

supplementary

Correctness of Kruskal and Prim (Optional)

Kruskal's algorithm: at every step, choose a minimum weighted edge (u, v) that does not form a cycle $\Longrightarrow u$ and v belong to two different trees T_1 and T_2. Set $S = T_1$ in Theorem, so after adding (u, v), the set of edges still belong to a MST. Keep repeating this till $n - 1$ edges, therefore final tree is MST.

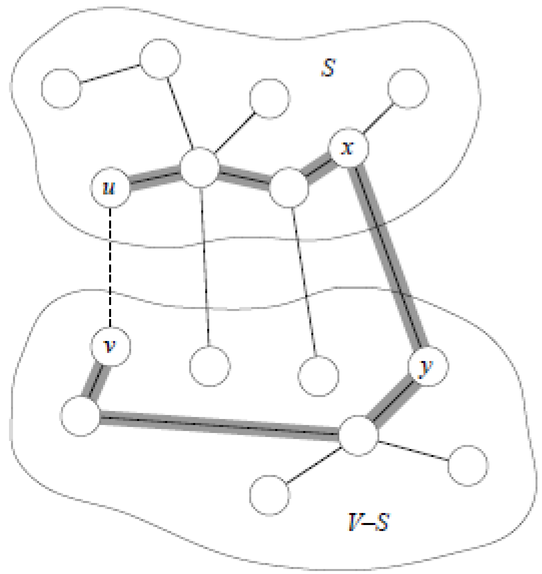
"Generic" MST algorithm:

- **Start with $A = \emptyset$. Let $G_A = (V, A)$ - this is a forest.**
- **While A is not a spanning tree, take any connected component S in G_A, find an edge with minimum weight connecting S to some other component in G_A. Add this edge to A.**

Proof of Theorem:

Let T be a MST containing A. If T contains (u, v), done.

Assume T does not contain (u, v). We construct another MST T' that contains $A \cup (u, v)$.



$T' = (T - \{(x, y)\}) \cup \{(u, v)\}$ **is a spanning tree and**

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$$

so T' must be a MST.

Learning Takeaway

- **Greedy algorithm in which you take the best action at each step can produce the overall optimal solution. But this is not always the case.**

- **Note the difference between a shortest path tree SPT and a minimum spanning tree MST.**

- **To find SPT:**
  - **weighted graph with non-negative weights: Dijkstra's Algorithm.**
  - **Unweighted graph: BFS**

- **To find MST: Kruskal and Prim's algorithms**