EE2008: Data Structures and Algorithms

Searching and Greedy Algorithms

Prof Huang Guangbin

S2.1-B2-06

egbhuang@ntu.edu.sg

Outline of Coverage

- **Searching**
  - **Different types of search algorithms**
- **Greedy Algorithms to find**
  - **Minimum spanning trees**
  - **Shortest paths**

SEARCHING

What is Searching?

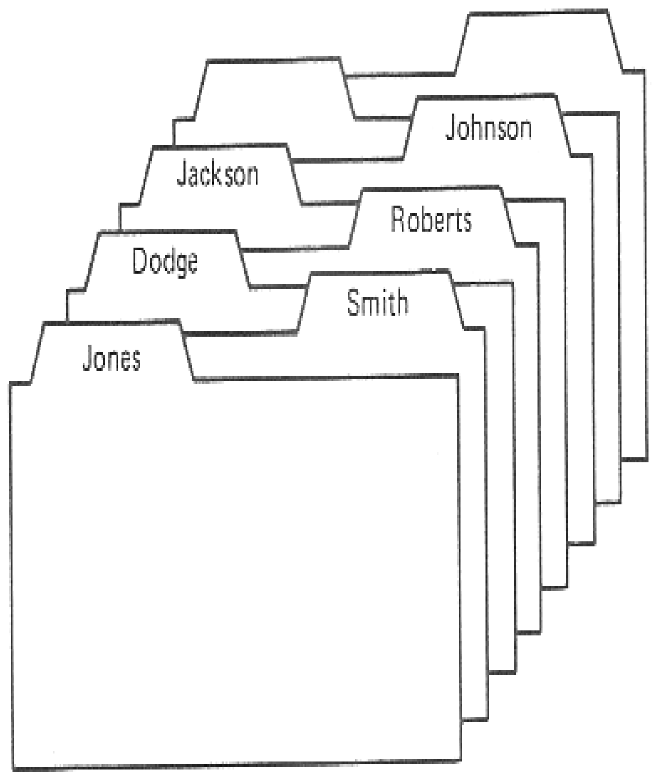
- **Searching**
  - retrieving information from a large amount of previously stored information

- **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

A/P Huang Guangbin

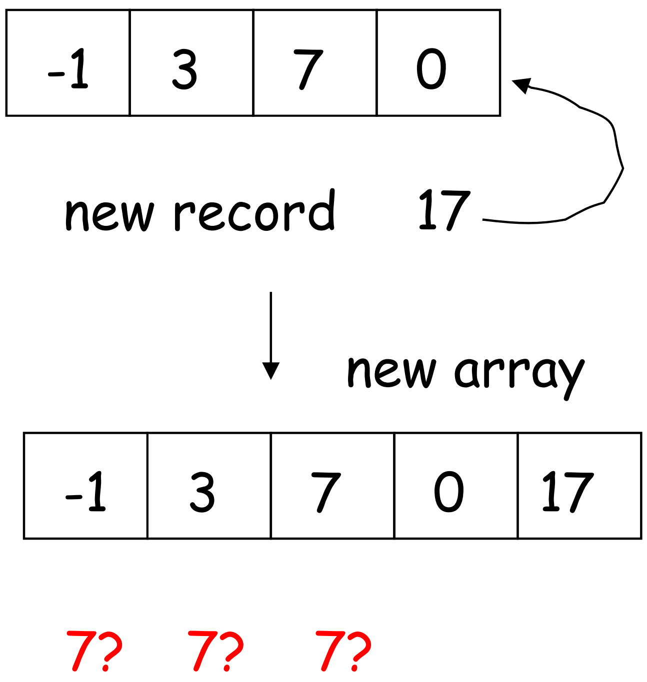
Searching Methods

- **Elementary searching methods**
  - **Sequential (Linear) search**
  - **Binary search**
- **Text Searching Algorithms**
- **Graph Search Algorithms**

Sequential 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,n,key) {
    for (k = 0; k < n) {
        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

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] ≤ L[2] ≤ … ≤ L[n-1] ≤ 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]

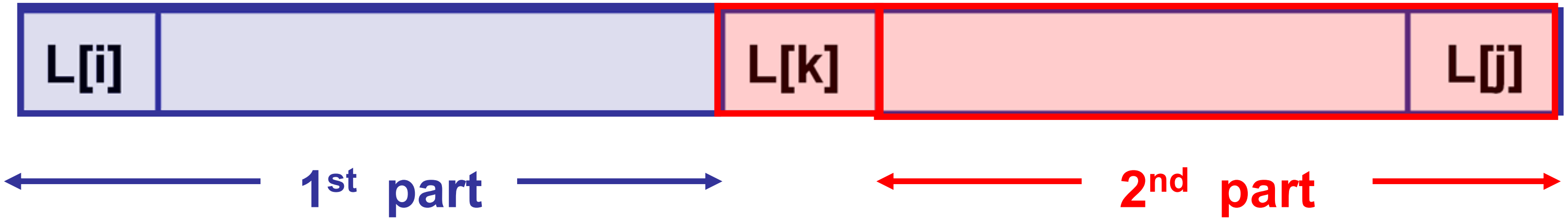
k = $\left\lfloor \frac{1+5}{2} \right\rfloor$ = 3

- 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=28, i=1, j=8

Loop 1

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

Key=28, 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=28, 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=28, k=5

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

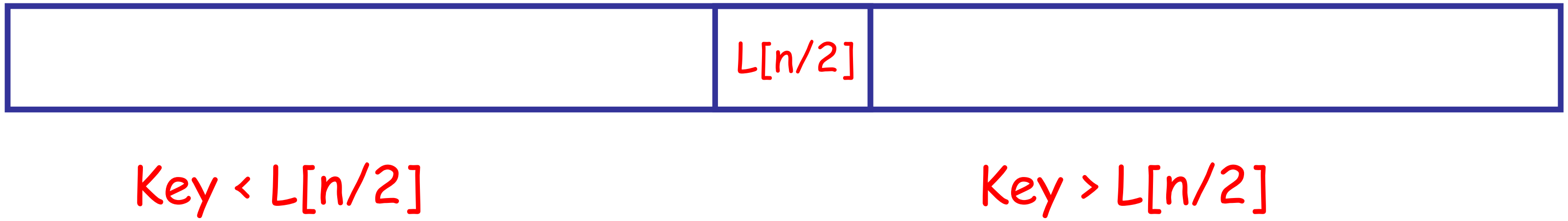
Loop 3

	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



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

$$
\begin{aligned}
&= b + T(n/2)\\
&= b + [b + T(n/4)] = 2b + T(n/4)\\
&= 2b + [b + T(n/8)] = 3b + T(n/8)\\
&= \ldots\\
&= kb + T(n/2^k)\\
&= \ldots\\
&= b^*\log(n) + T(1)\\
&= b^*\log(n) + b\\
&= b(\log(n)+1). \text{ Thus, } O(\log(n))
\end{aligned}
$$

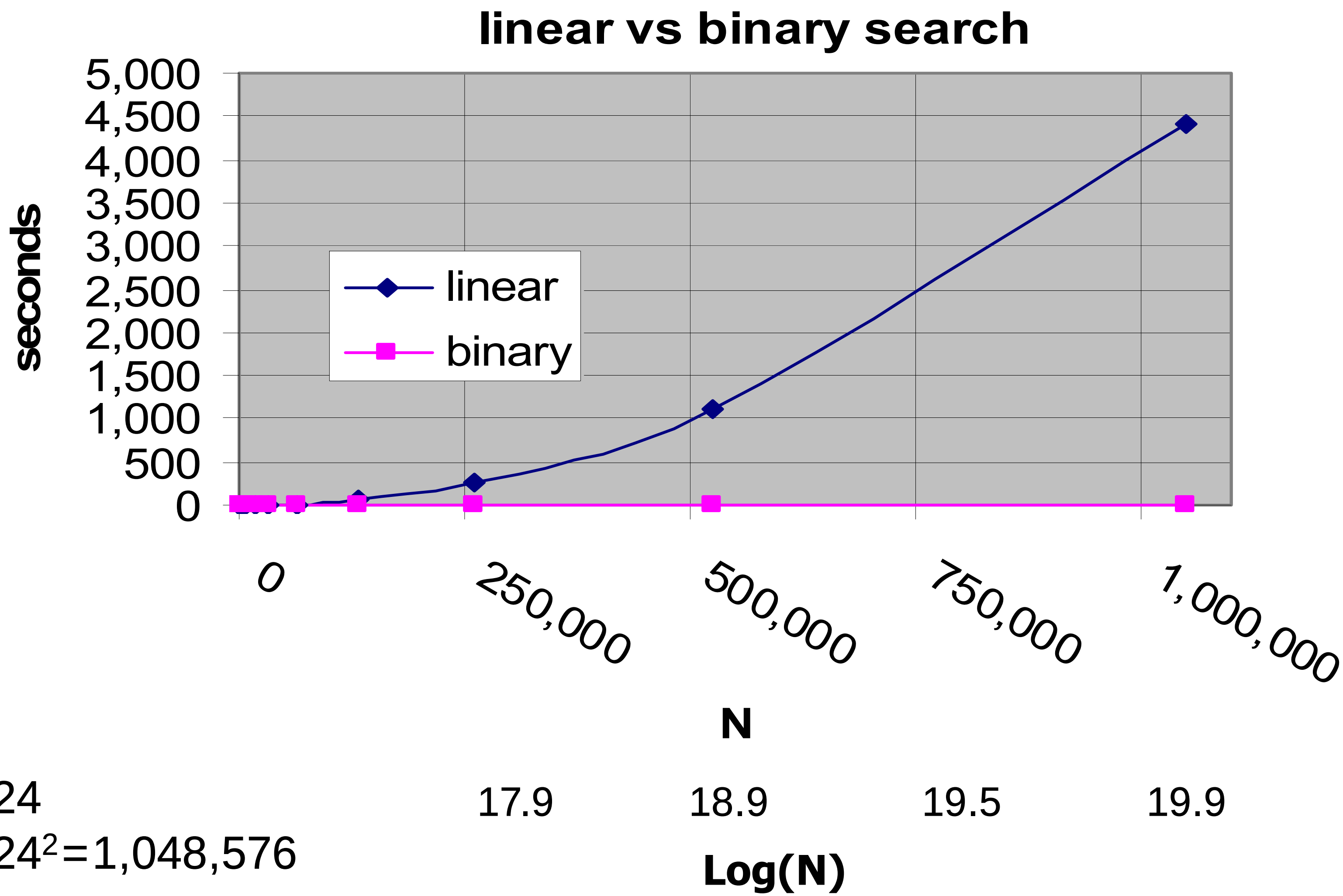
$n/2^k = 1 \Rightarrow 2^k = n$

$\Rightarrow k = \log(n)$

Sequential Search vs Binary Search

Sequential (linear) search : O(n)

Binary search : O(log(n))



$2^1=2$

$2^2=4$

$2^4=16$

$2^8=256$

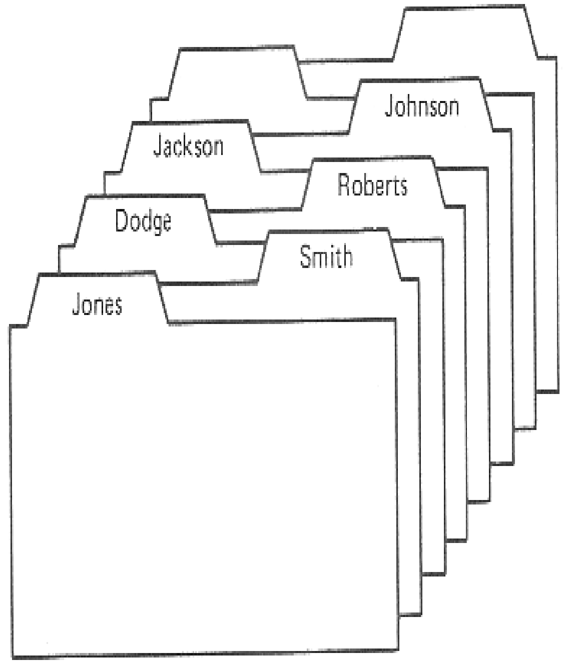
$2^{10}=1024$

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

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

Searching in Multi-dimensional Info

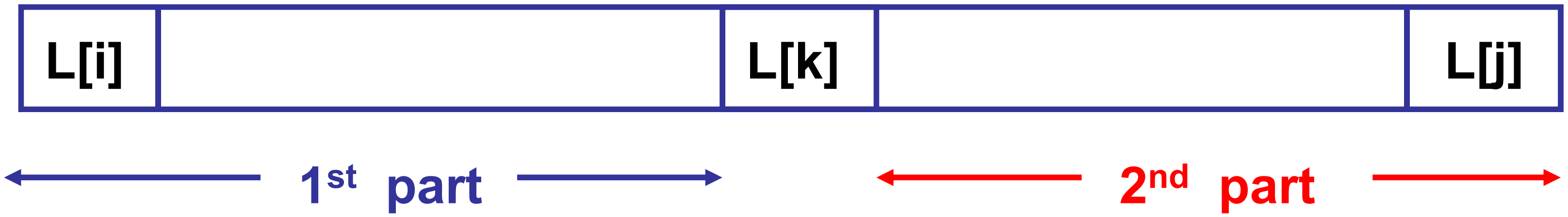
-1	3	7	0	17



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

Golden Section Search




```
golden_section_search(L,i,j,key) {
    while (i ≤ j) {
        k = i + 0.618(j - i)// golden section point
                                    // instead of 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
}
```

Sorting + Searching

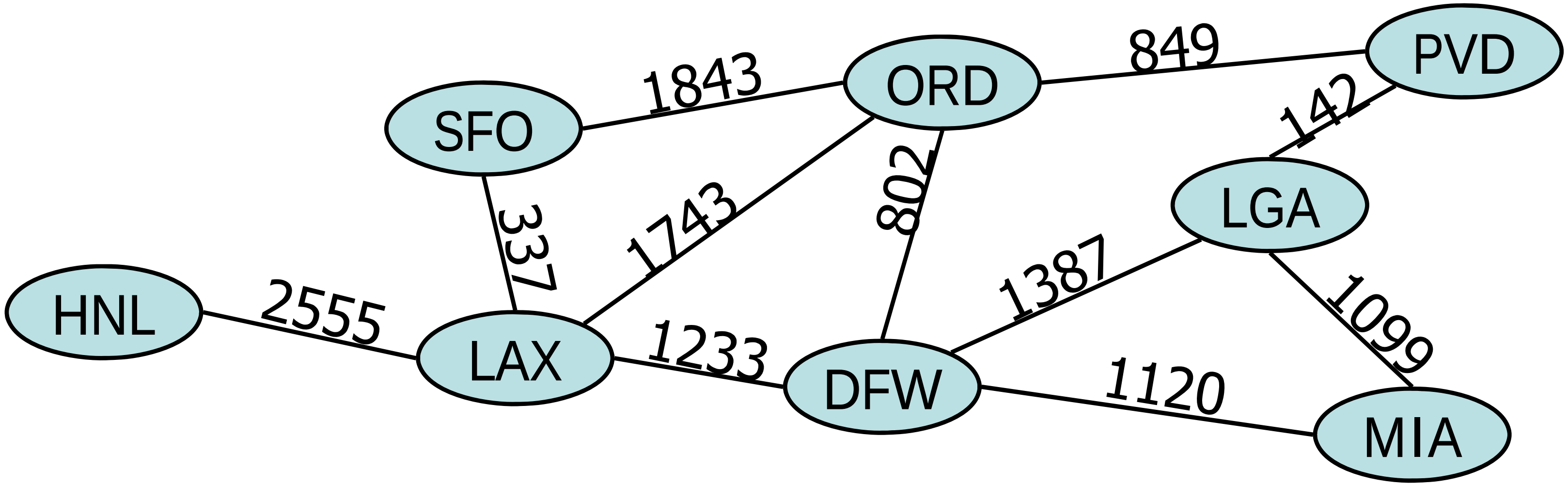
- Worst case time-complexity of Sorting+searching, usually: O($n \log n$) or above, but linear search is O(n)
  - ☞ Worst case time-complexity of Sorting, usually:
    - ✓ mergesort: O($n \log n$)
    - ✓ quicksort: O(n^2)
    - ✓ heapsort: O($n \log n$)
  - ☞ Worst case time complexity of Binary search O($\log n$)
- Then why do we need to have sorting + searching?

- Binary search, binary search tree same?

Graphs

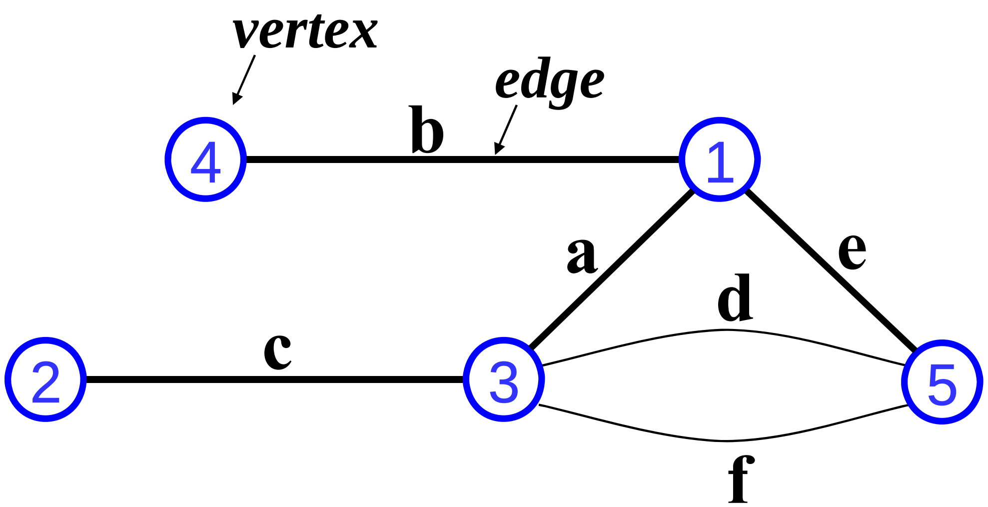
Graph

- 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



Undirected Graph

- 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}

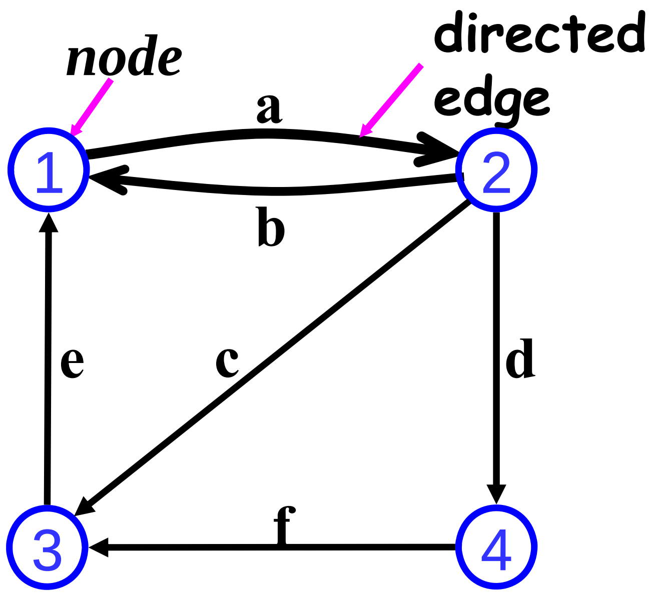
E = {a,b,c,d,e,f}

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

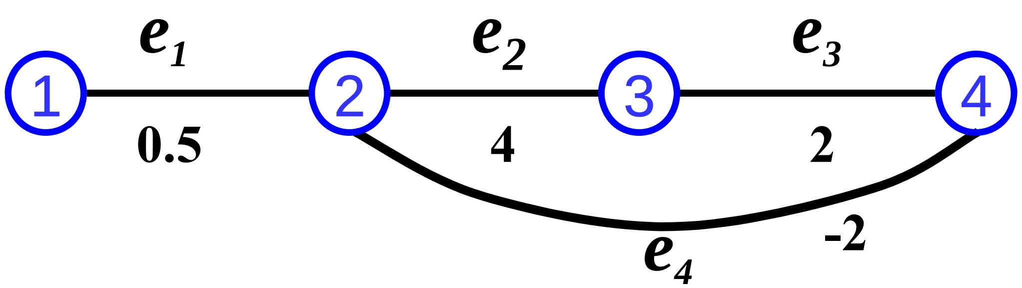
- A directed graph is a graph containing uni-directional edges
  - each edge is associated with an ordered pair of vertices
    - if e is a directed edge connecting vertices u & v, then we write e = (u,v)
      - first vertex u is called the tail
      - second vertex v is called the head



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

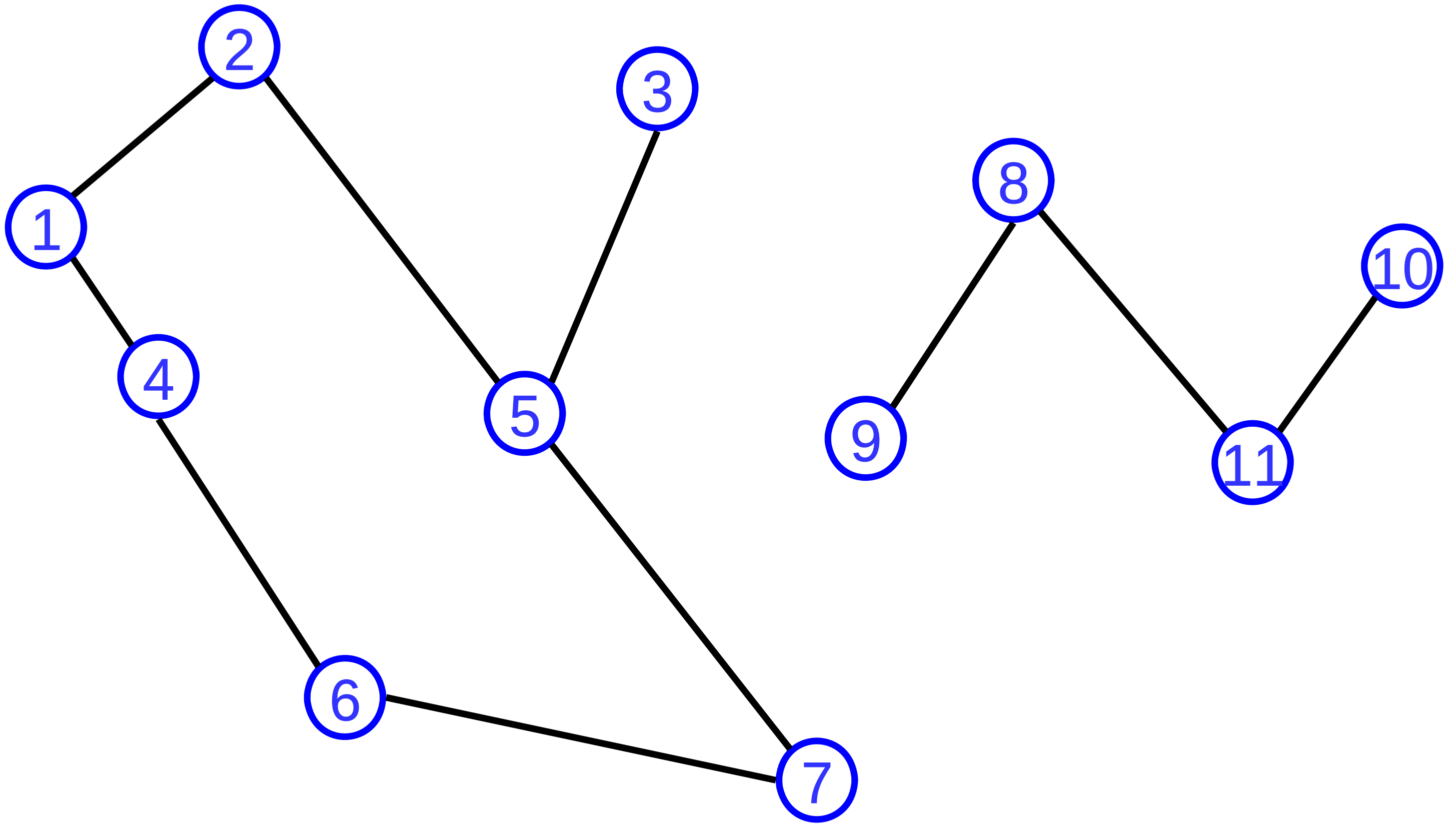
Weighted Graphs

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



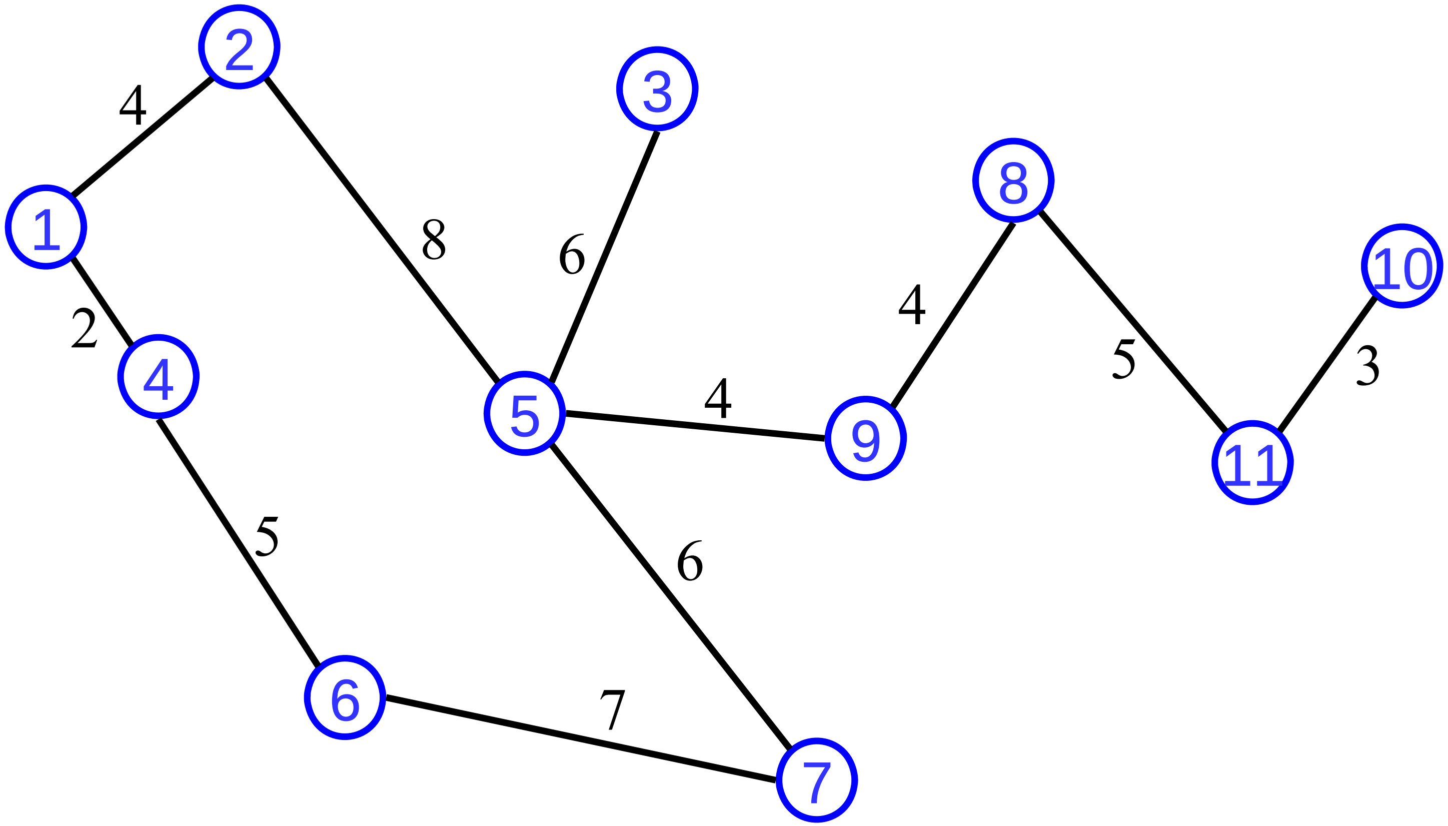
- The actual meanings of the numbers depend on the application. In general they may be positive or negative.

Applications—Communication Network



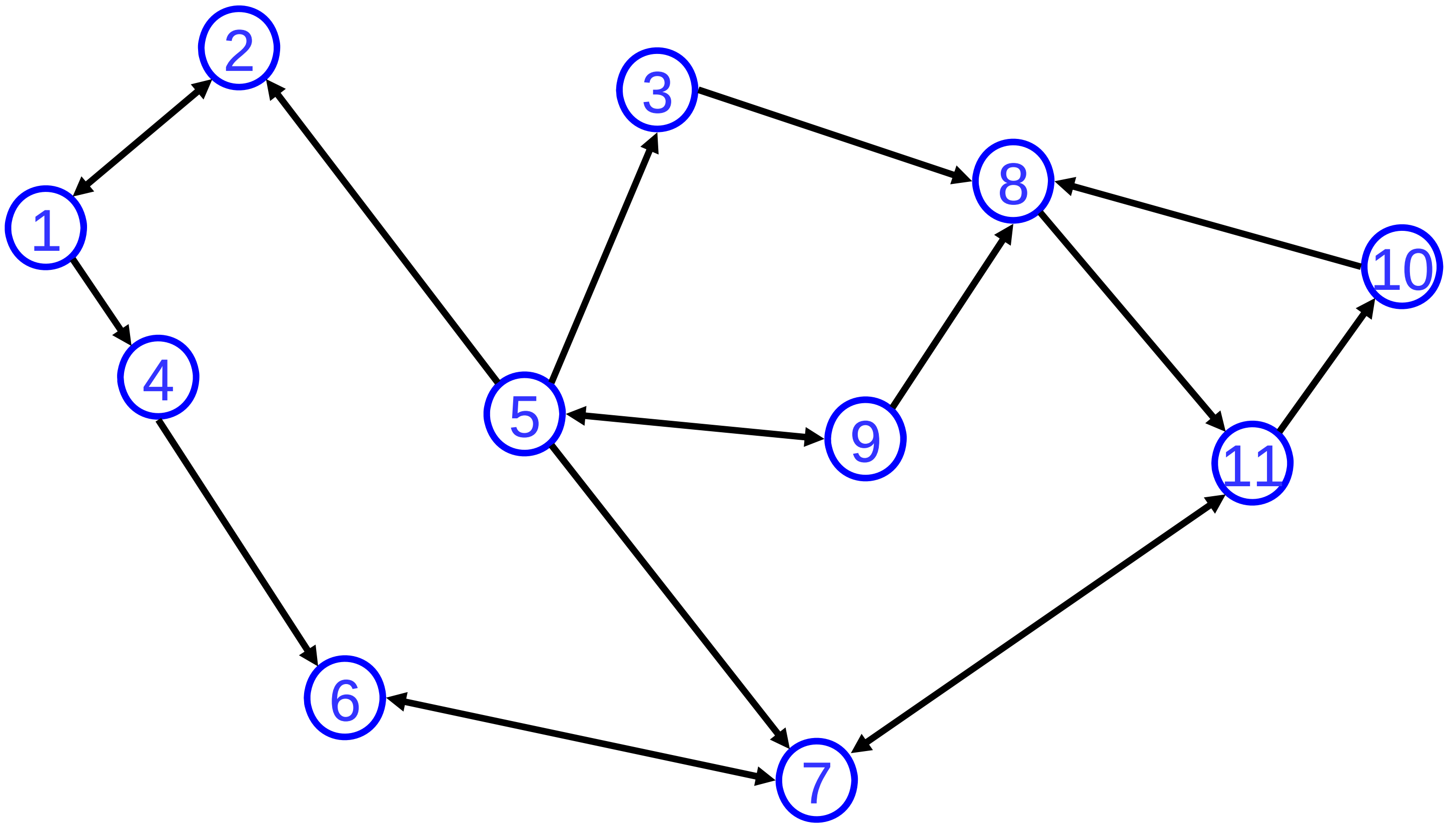
- **Vertex = city, edge = communication link.**

Driving Distance/Time Map



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

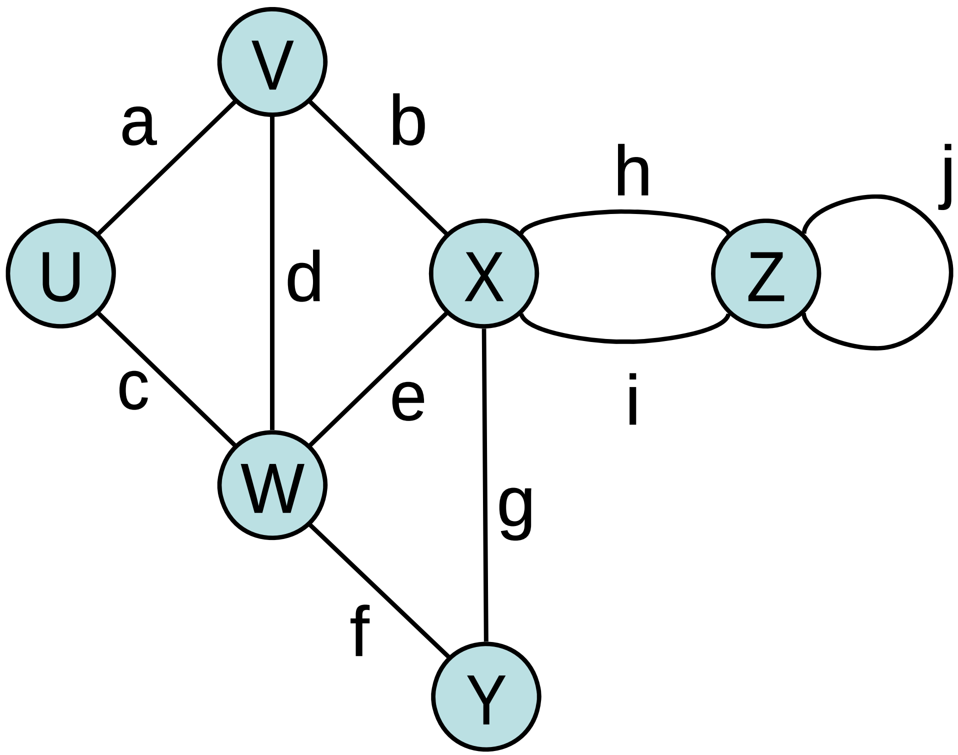
Street Map



- **Some streets are one way.**

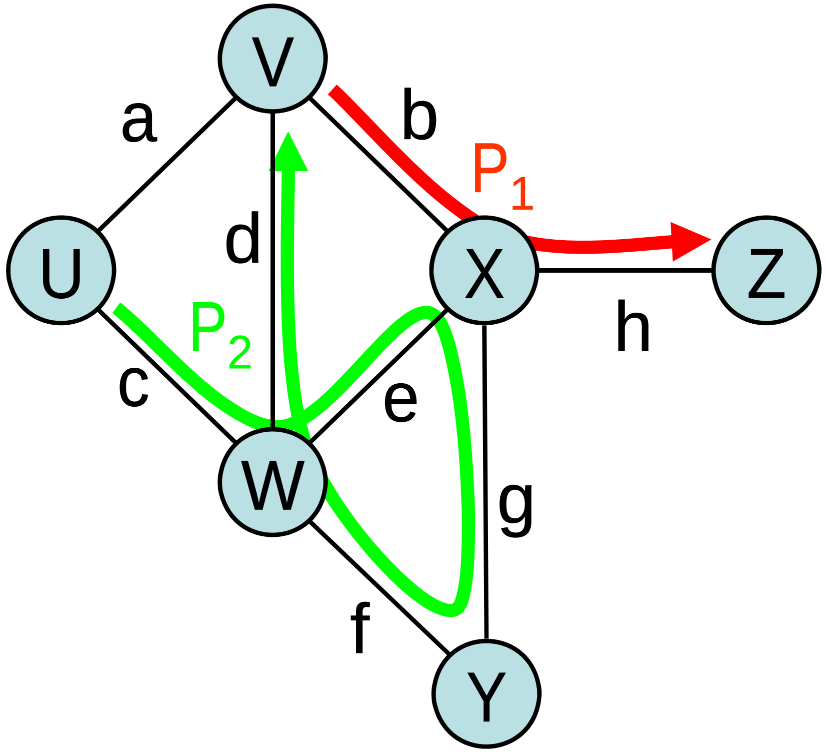
Terminology

- 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
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



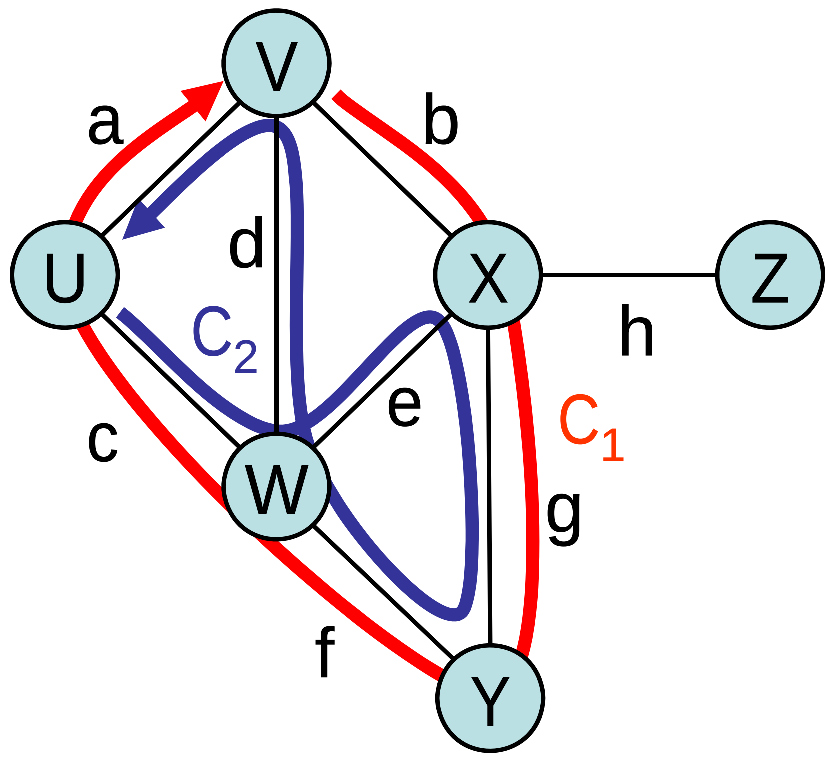
Terminology (cont.)

- **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



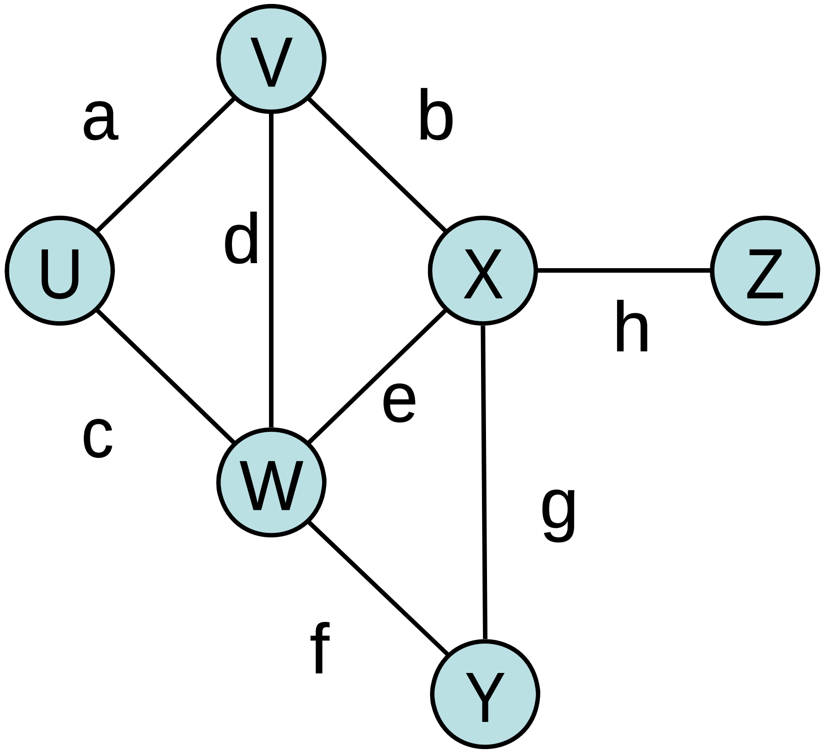
Terminology (cont.)

- **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

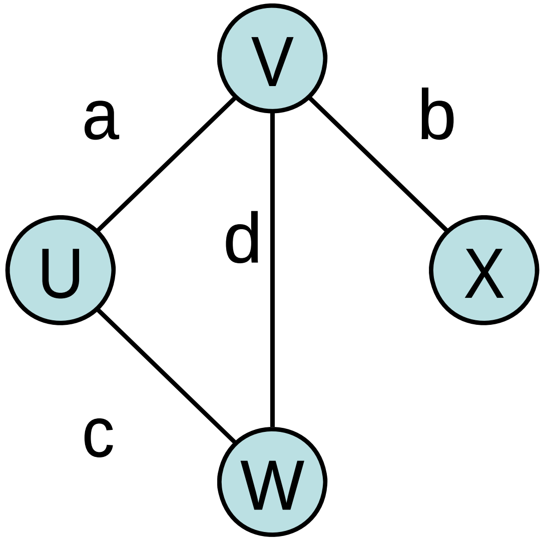


Subgraphs

- A graph *G'* is a **subgraph** of *G* if all its vertices and edges are in *G*.



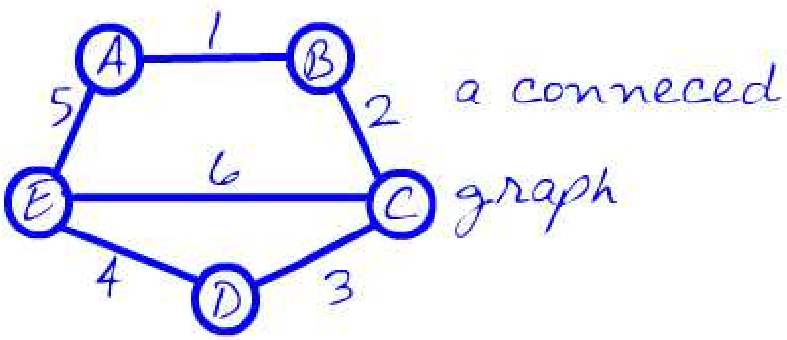
Graph G

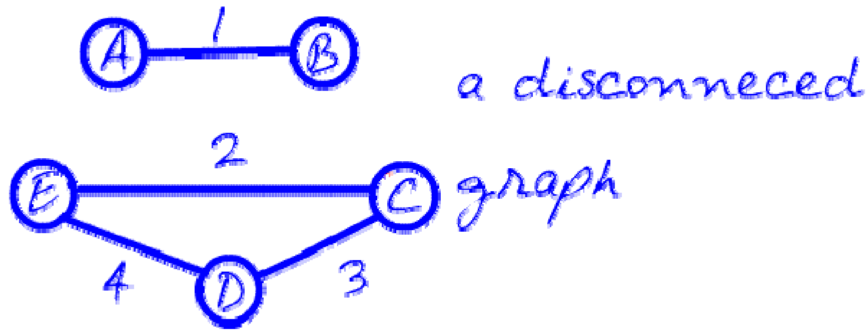


Subgraph of G

Connectivity

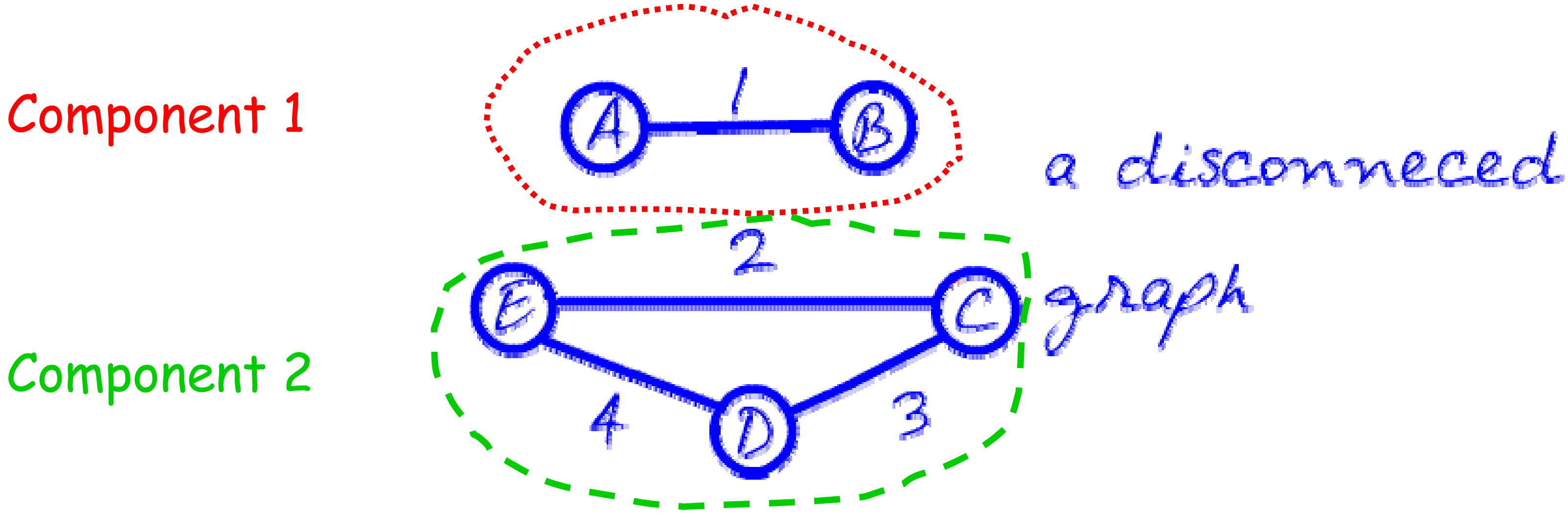
- **A graph is connected if there is a path joining every pair of distinct vertices; otherwise it is called disconnected.**





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.

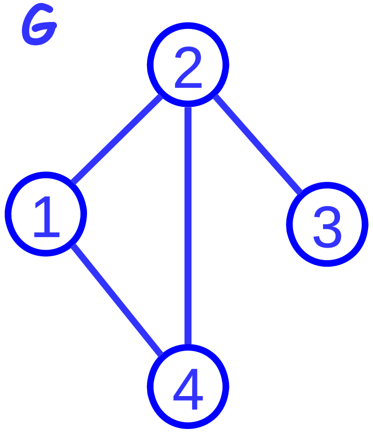


A graph of two components

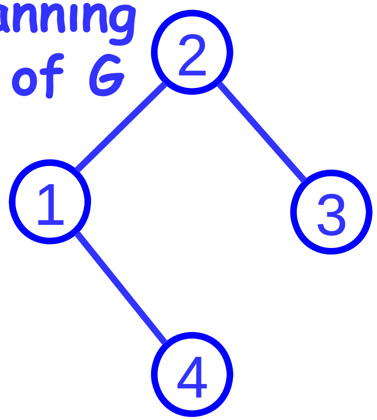
Trees

- 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

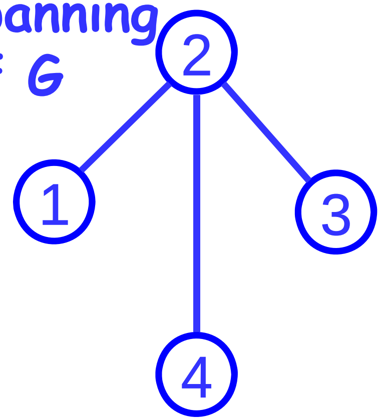
G



A spanning tree of G



Another spanning tree of G



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

- **Proof:**

Basis Step:

If n=1, the tree contains zero edge

If n=2, the tree contains one edge

Trees

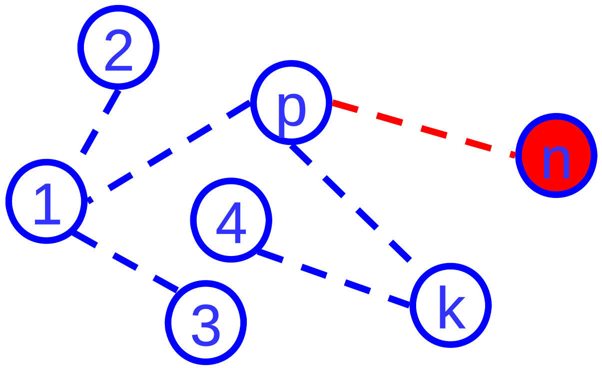
- **Why will any tree with n nodes contain n-1 edges?**

- **Proof:**

Inductive Step:

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

For n=k+1

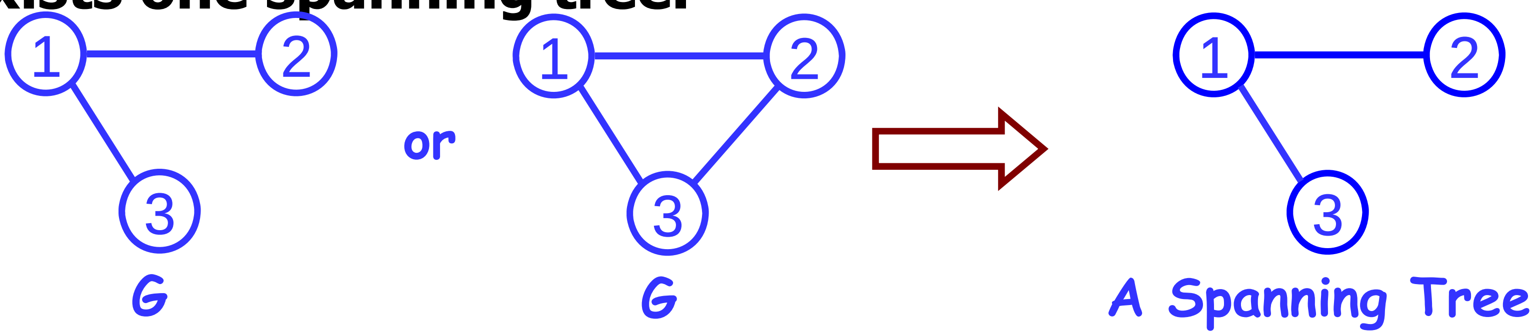


- **Why does every connected graph possess (at least) one spanning tree?**
- **Proof:**

Basis Step:

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.



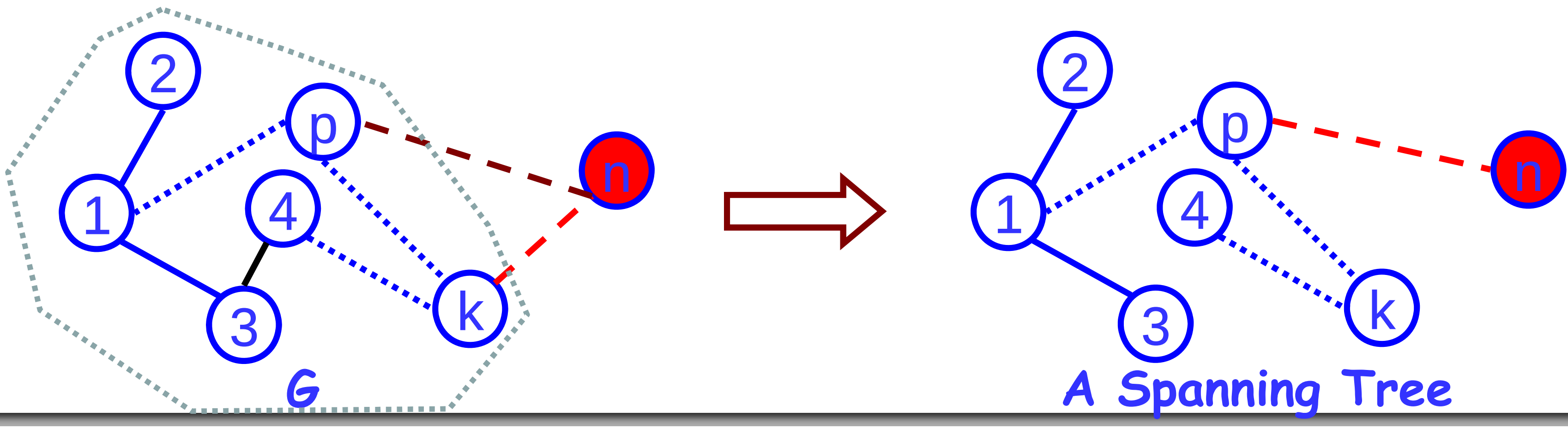
Why does every connected graph possess (at least) one spanning tree?

- **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

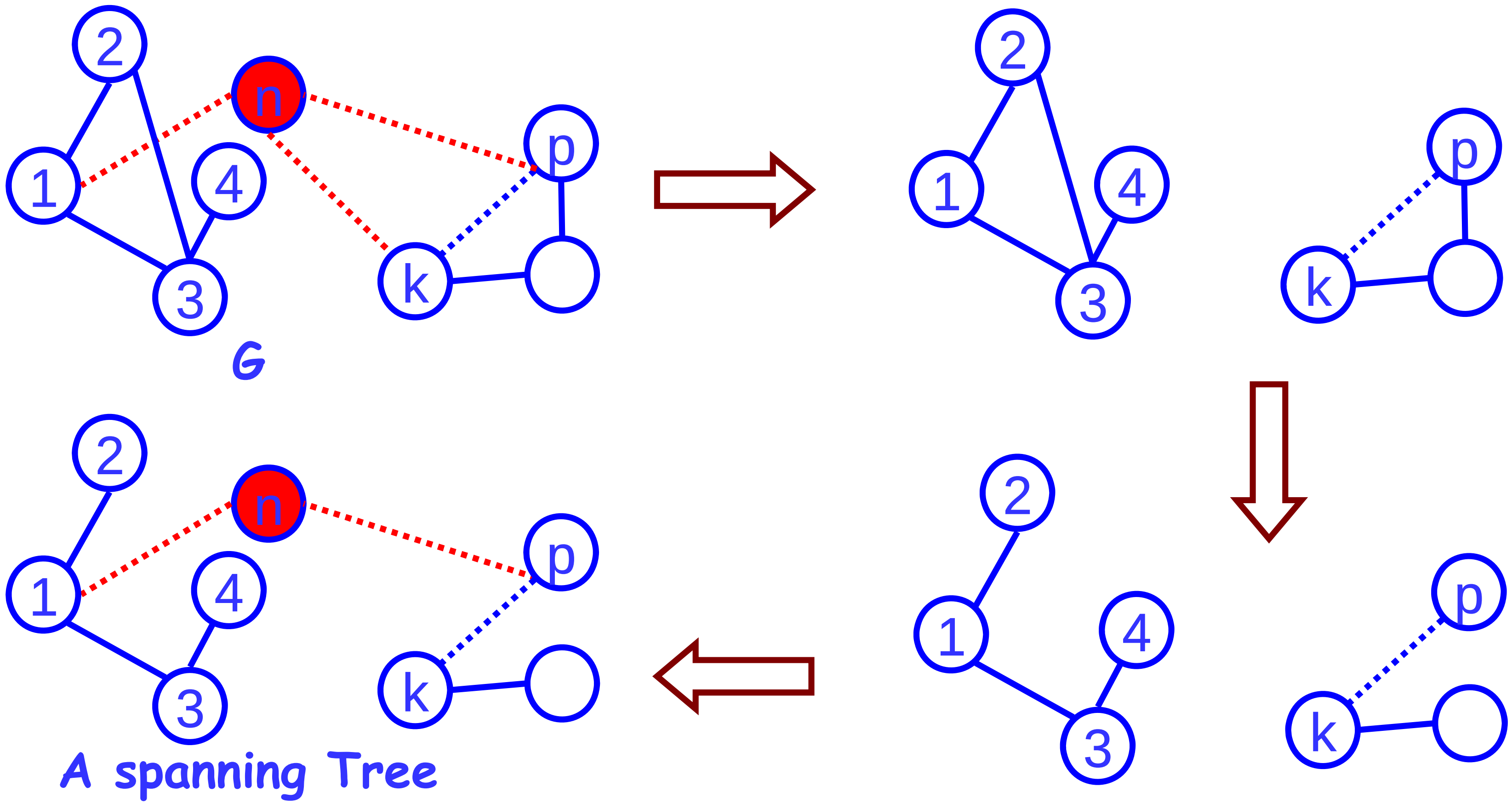


❑ Proof:

Inductive Step:

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

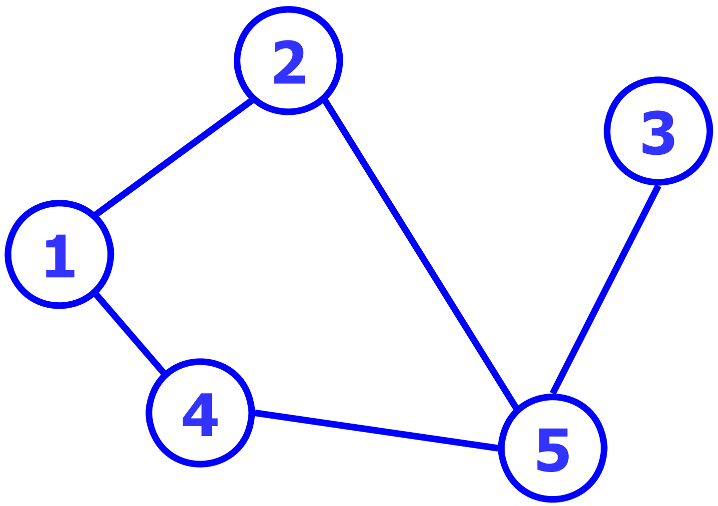
For for Graph G with n=k+1 vertices



Graph Representations

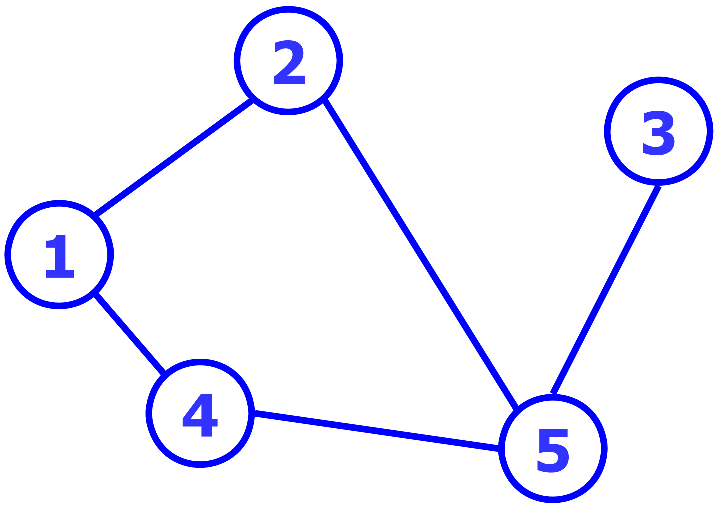
Adjacency Matrix

- 0/1 n x n matrix, where n = # of vertices
- A(i,j) = 1 iff (i,j) is an edge



	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

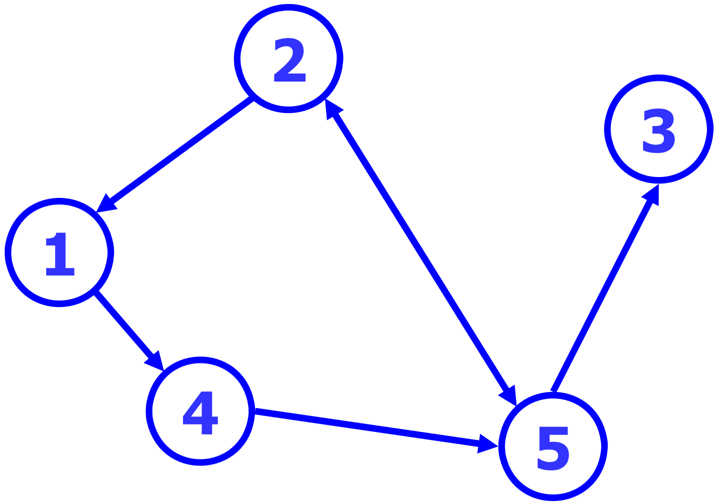
Adjacency Matrix Properties



	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 (Digraph)



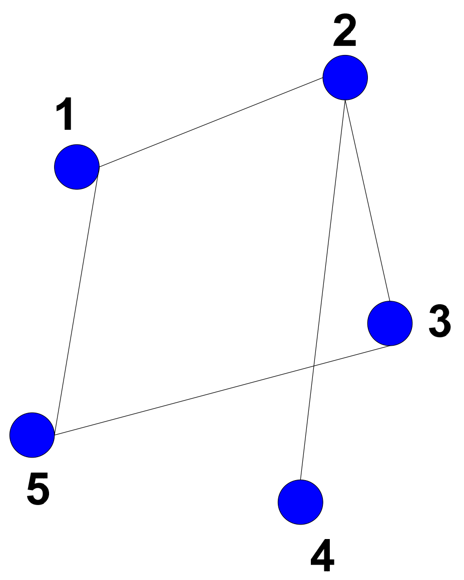
	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	1	1	0	0

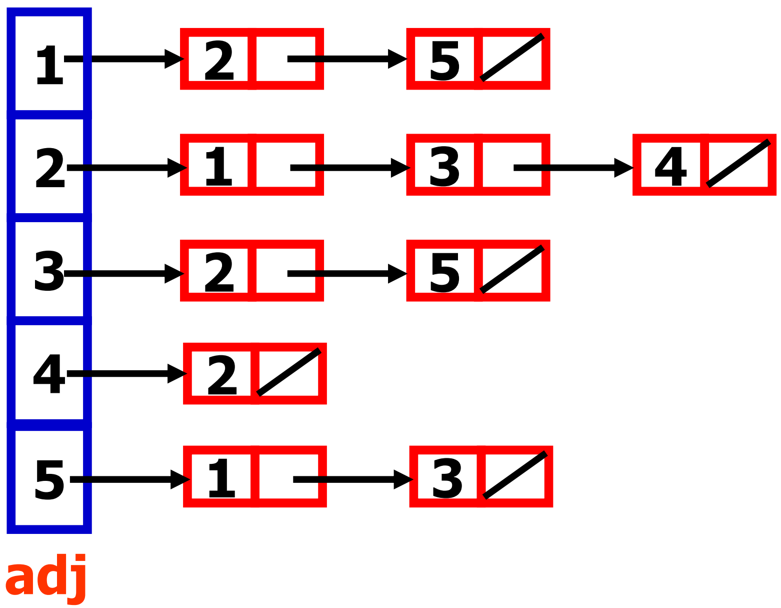
Adjacency Matrix

- 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.

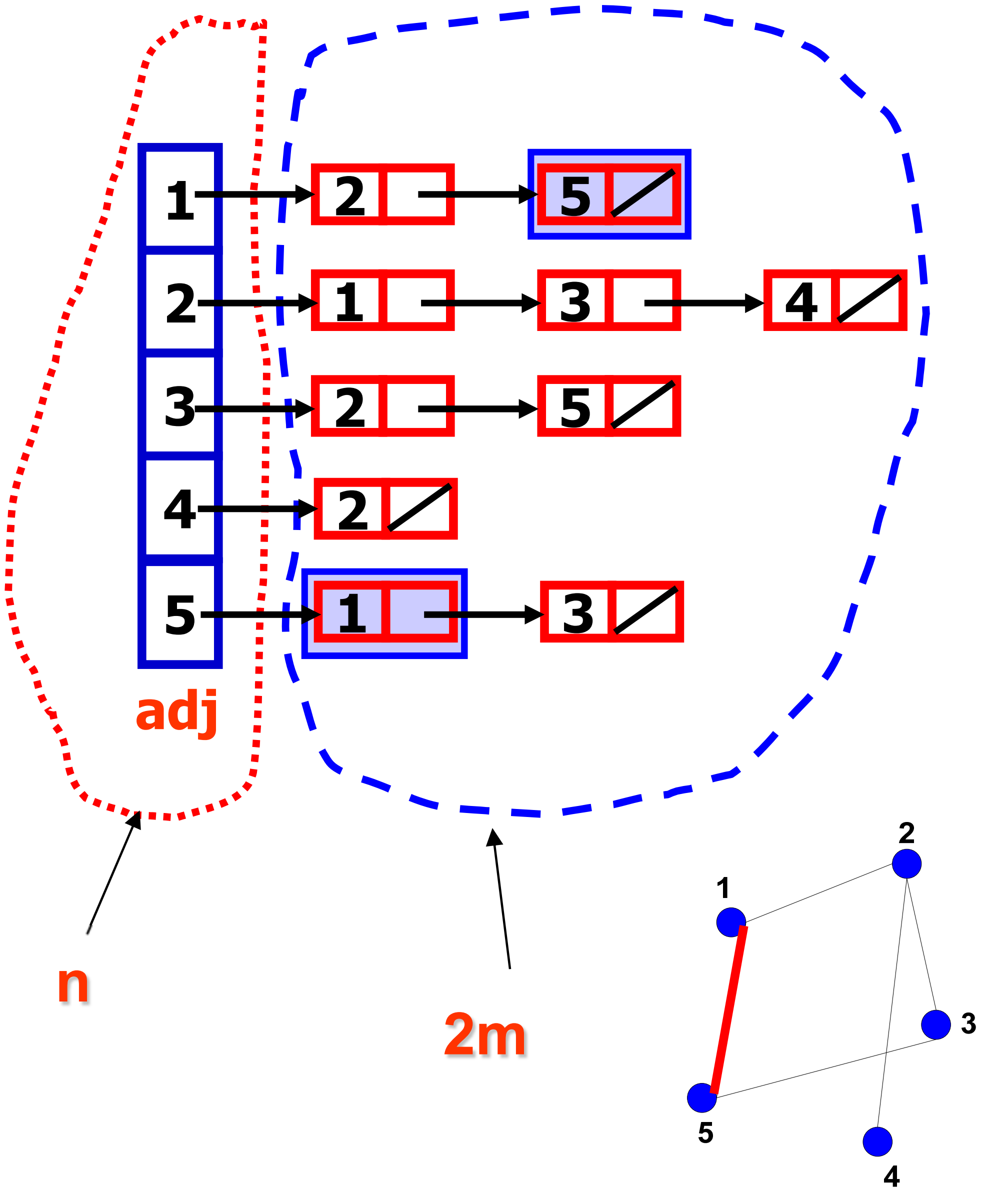
Adjacency Lists

- 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





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

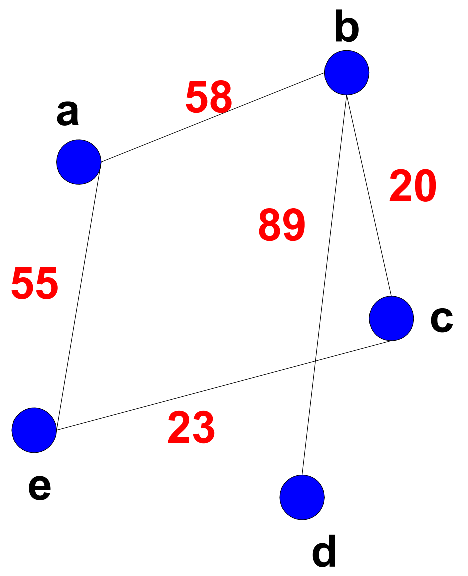
Space complexity = O(n+m)

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 Lists based algorithms is more efficient than Adj Matrix based algorithms?**

Weighted Graphs

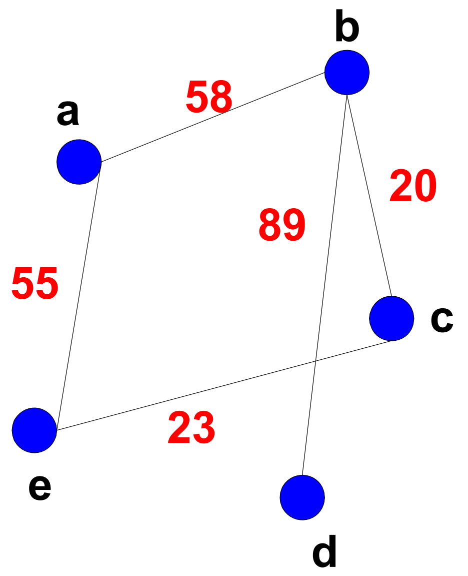
- Cost adjacency matrix
  - C(i,j) = cost of edge (i,j)
- Adjacency lists => each list element is a pair (adjacent vertex, edge weight)

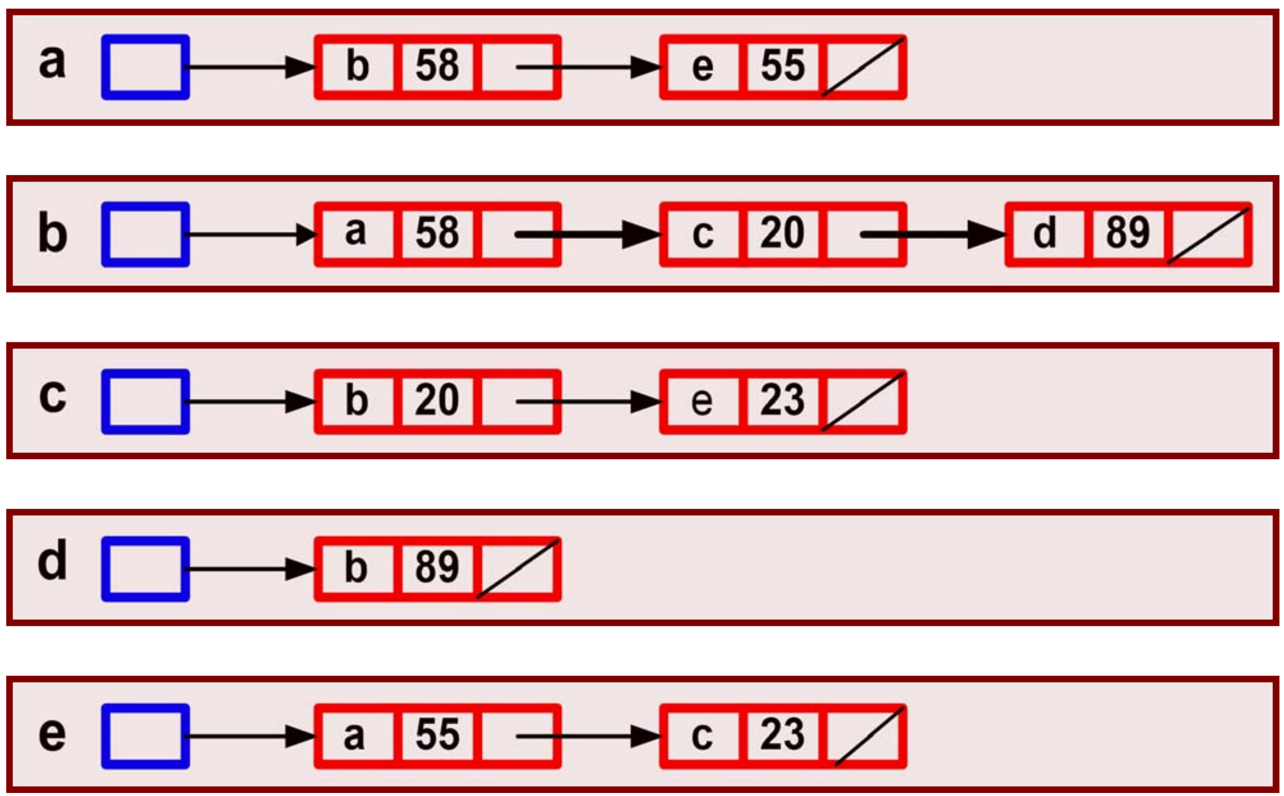


	a	b	c	d	e
a	0	58	0	0	55
b	58	0	20	89	0
c	0	20	0	0	23
d	0	89	0	0	0
e	55	0	23	0	0

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)

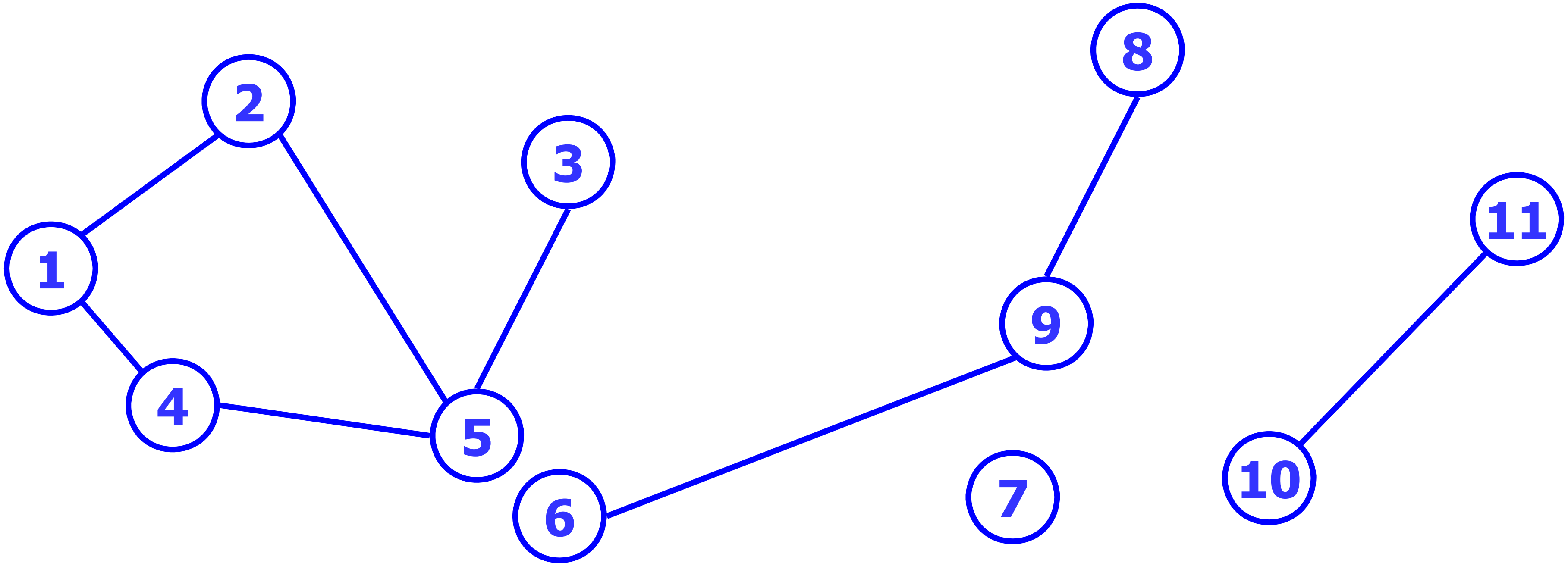




Graph Search Methods

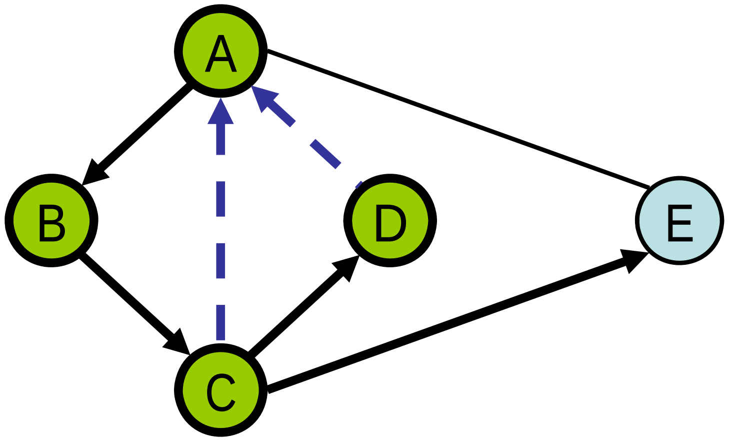
Graph Search Methods

- 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.

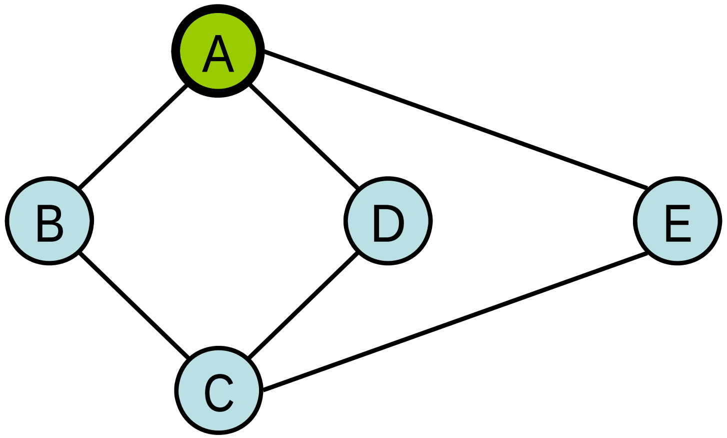


 A/P Huang Guangbin

Depth-First Search

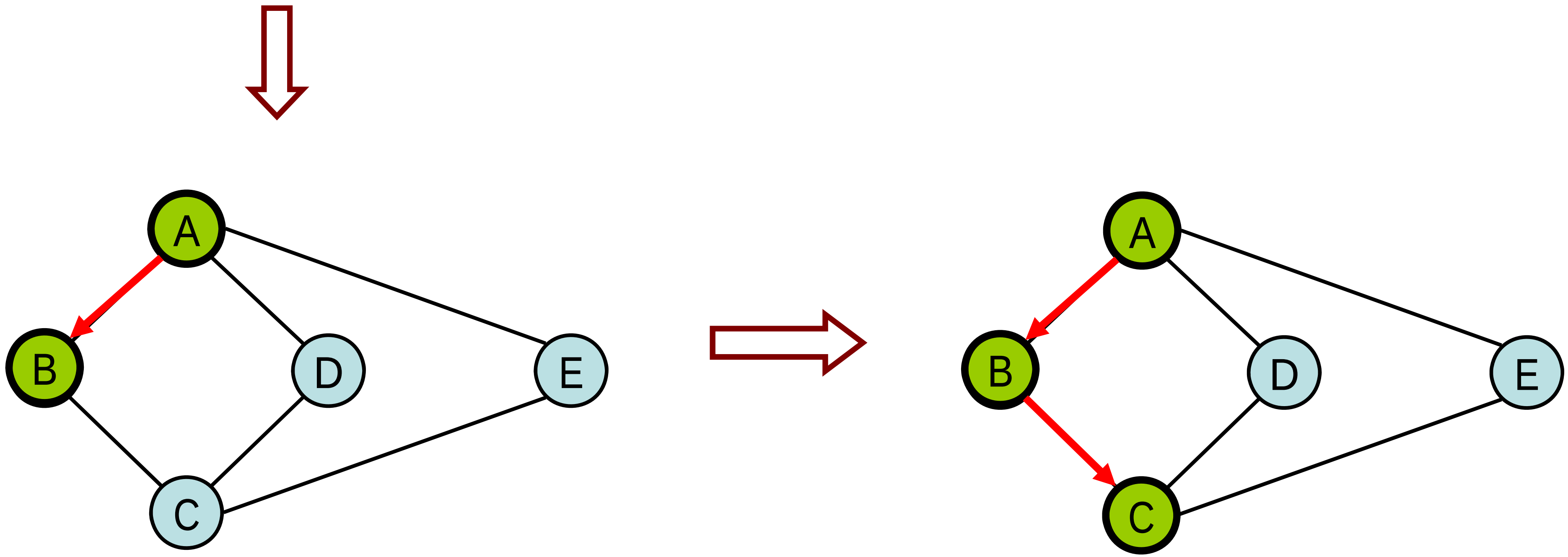


Example

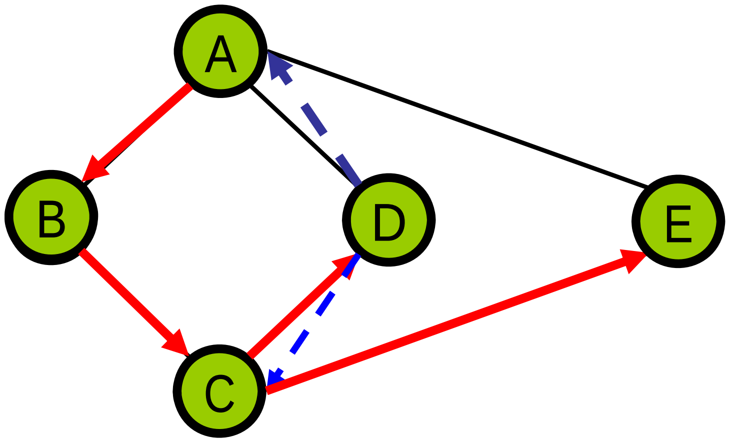


Depth-First-Search (v)

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

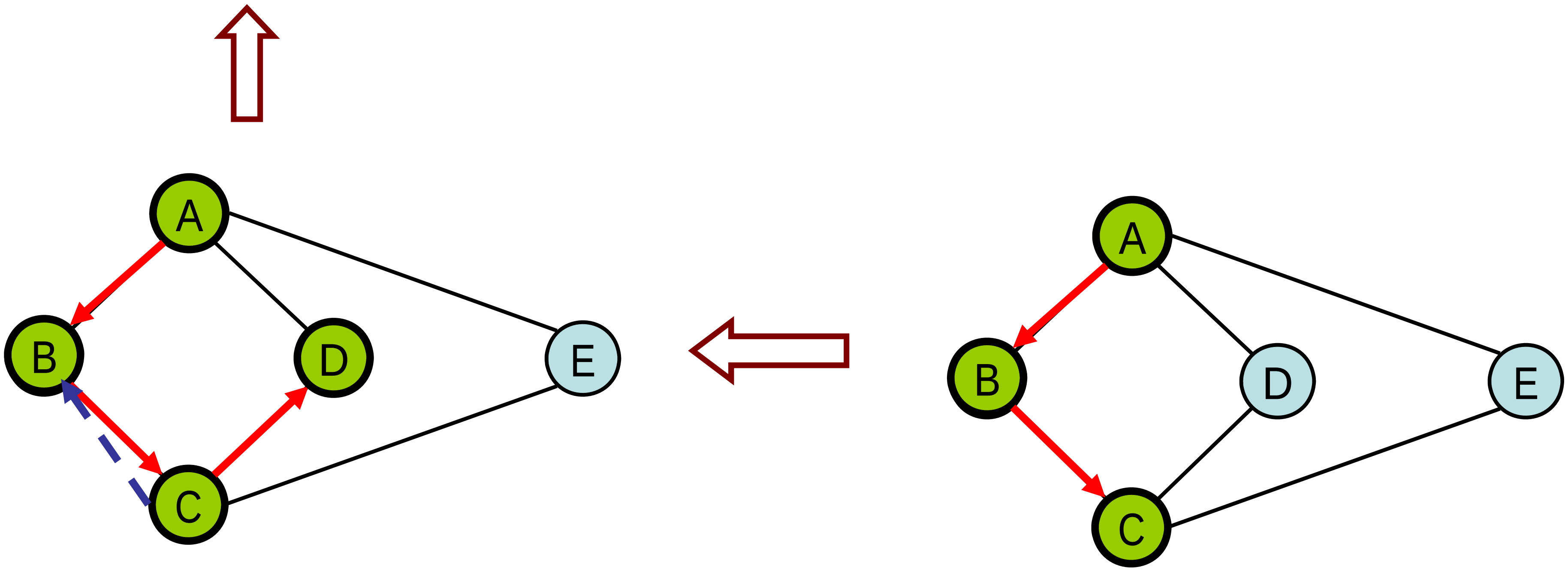


Example

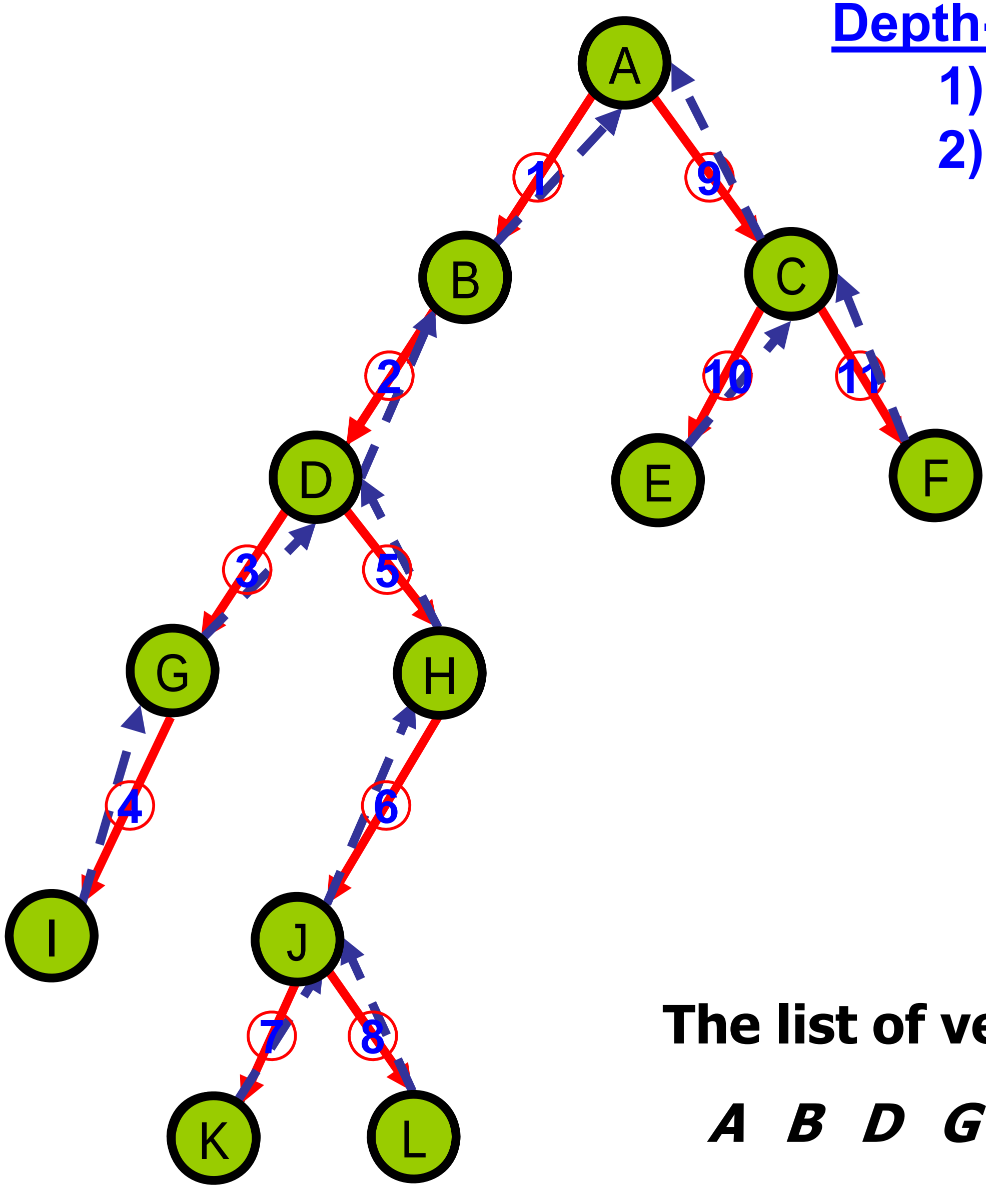


Depth-First-Search (v)

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



Example



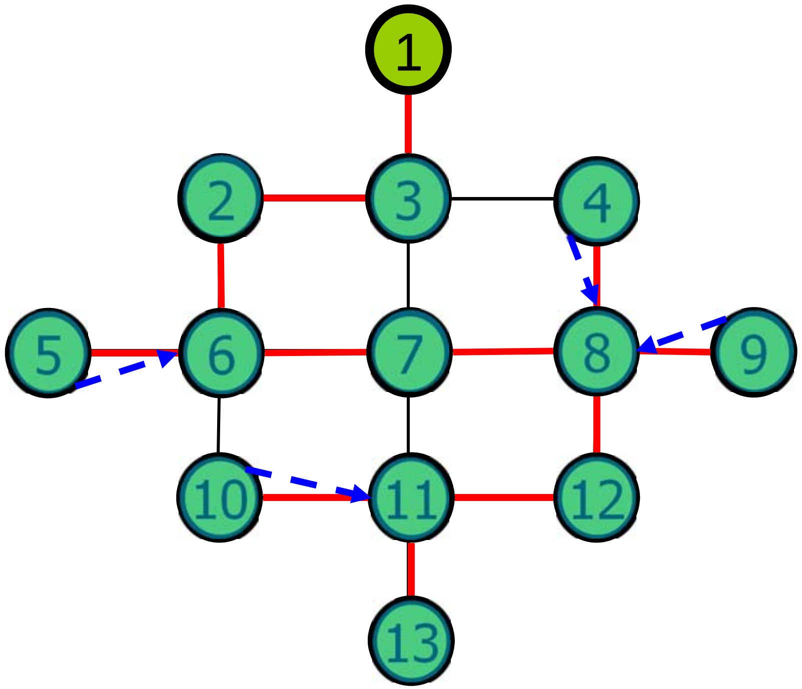
Depth-First-Search (v)

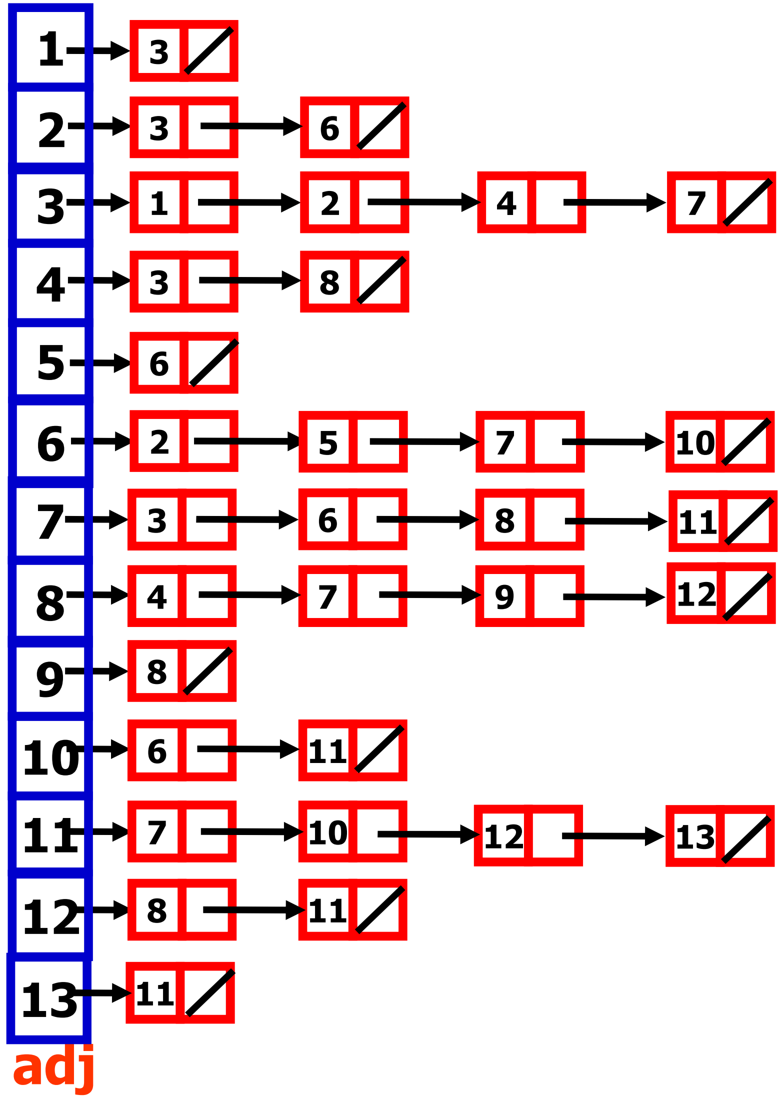
1) visit v, label v as visited.
2) For each unvisited vertex u adjacent to v, execute Depth-First-Search on u.

The list of vertices visited in order is:

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

Depth-First Search





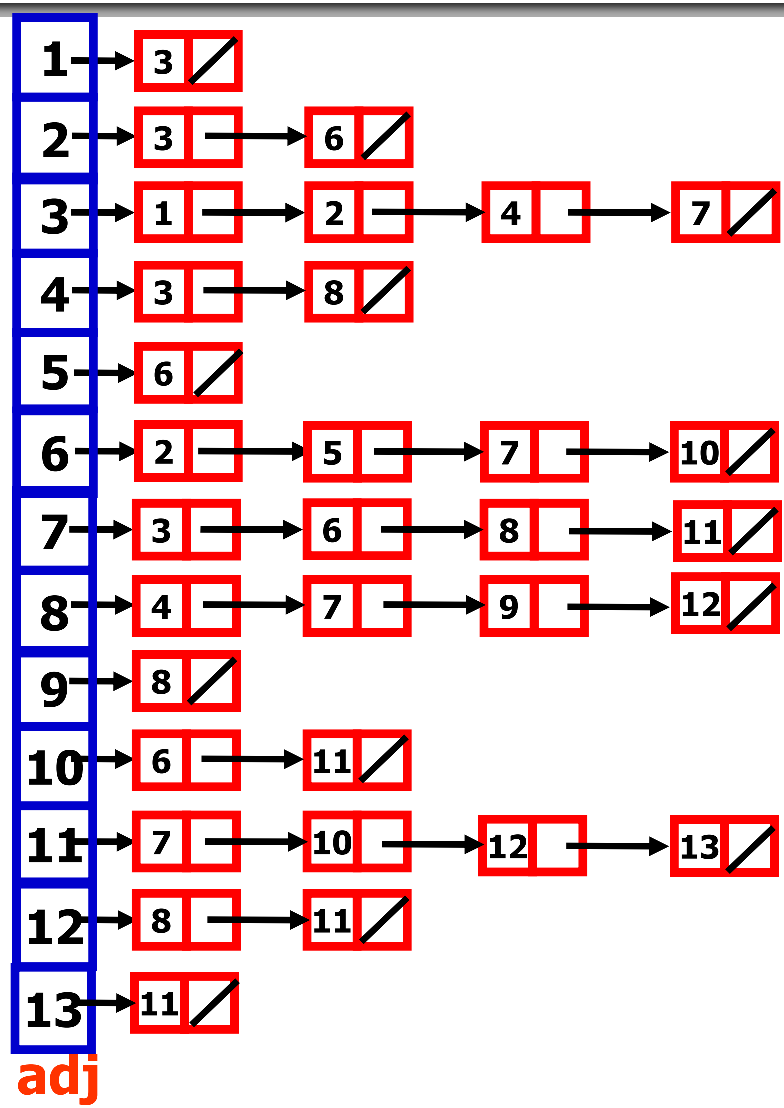
The list of vertices visited in order is:

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,s)
}

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



<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.

A/P Huang Guangbin

Depth-First Search

- This algorithm executes a depth-first search beginning at vertex *s* in a graph with vertices 1, ... , *n*
- 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,s)
}

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

DFS Time Complexity

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

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

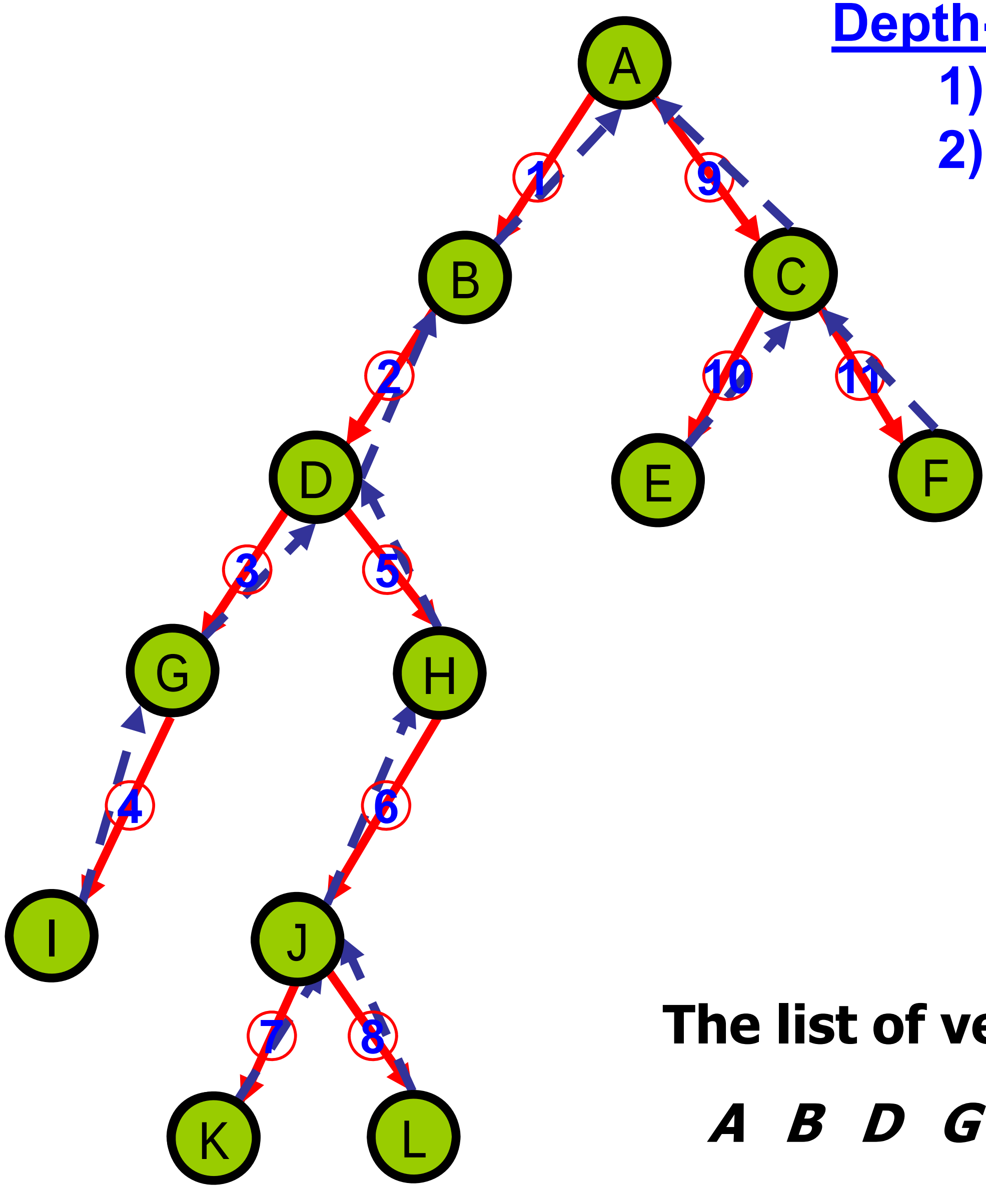
O(n) $<=k_1n$

Visit nodes in adjacency lists: O(m) $<=k_2m$

$O(n)+O(m) <= k_1n+k_2m <= \max(k_1, k_2) * (n+m)$

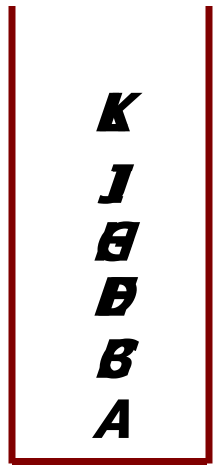
Overall time complexity = O(n+m)

Example



Depth-First-Search (v)

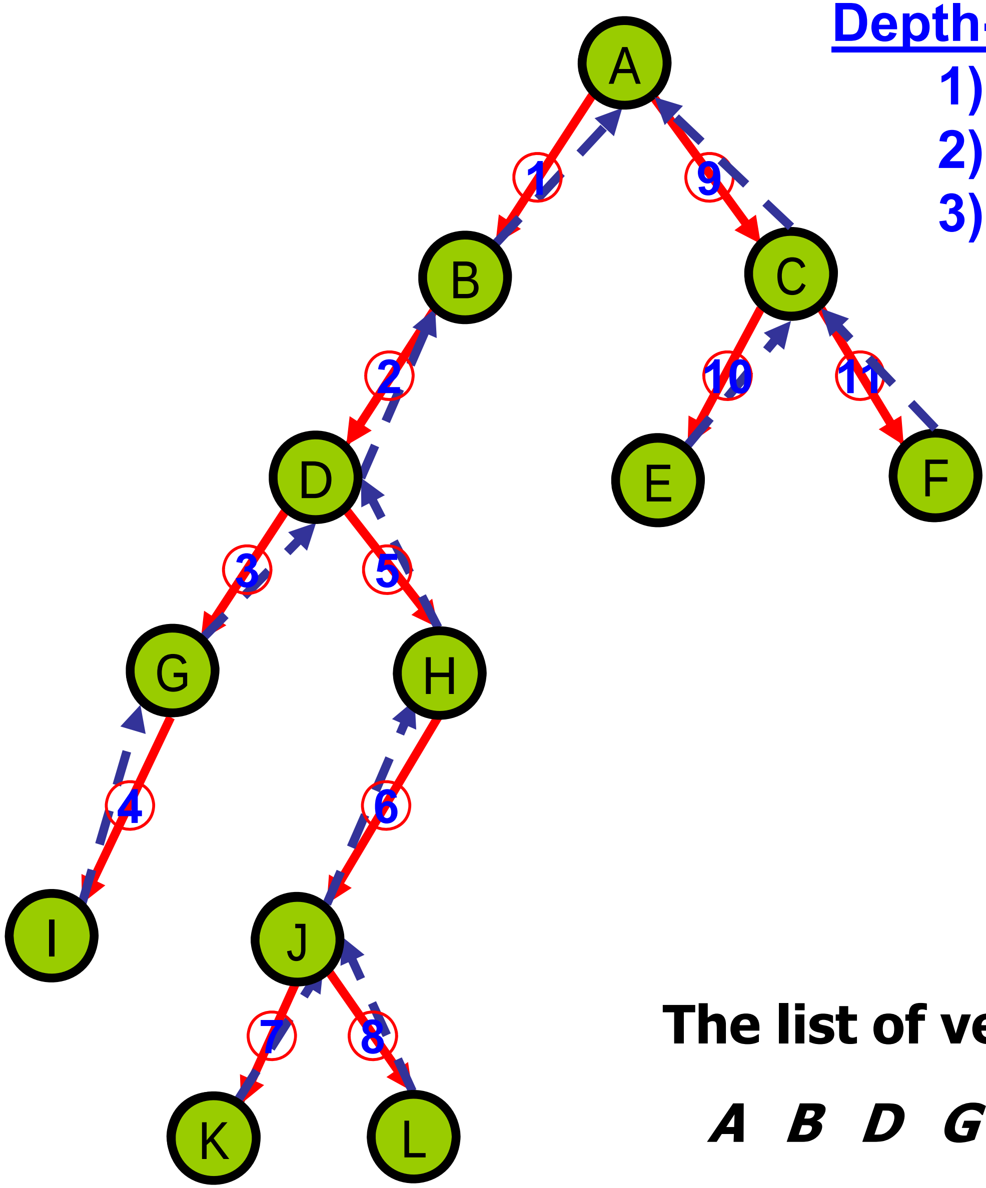
1) visit v, label v as visited.
2) For each unvisited vertex u adjacent to v, execute Depth-First-Search on u.



The list of vertices visited in order is:

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

Example



Depth-First-Search (adj, start)

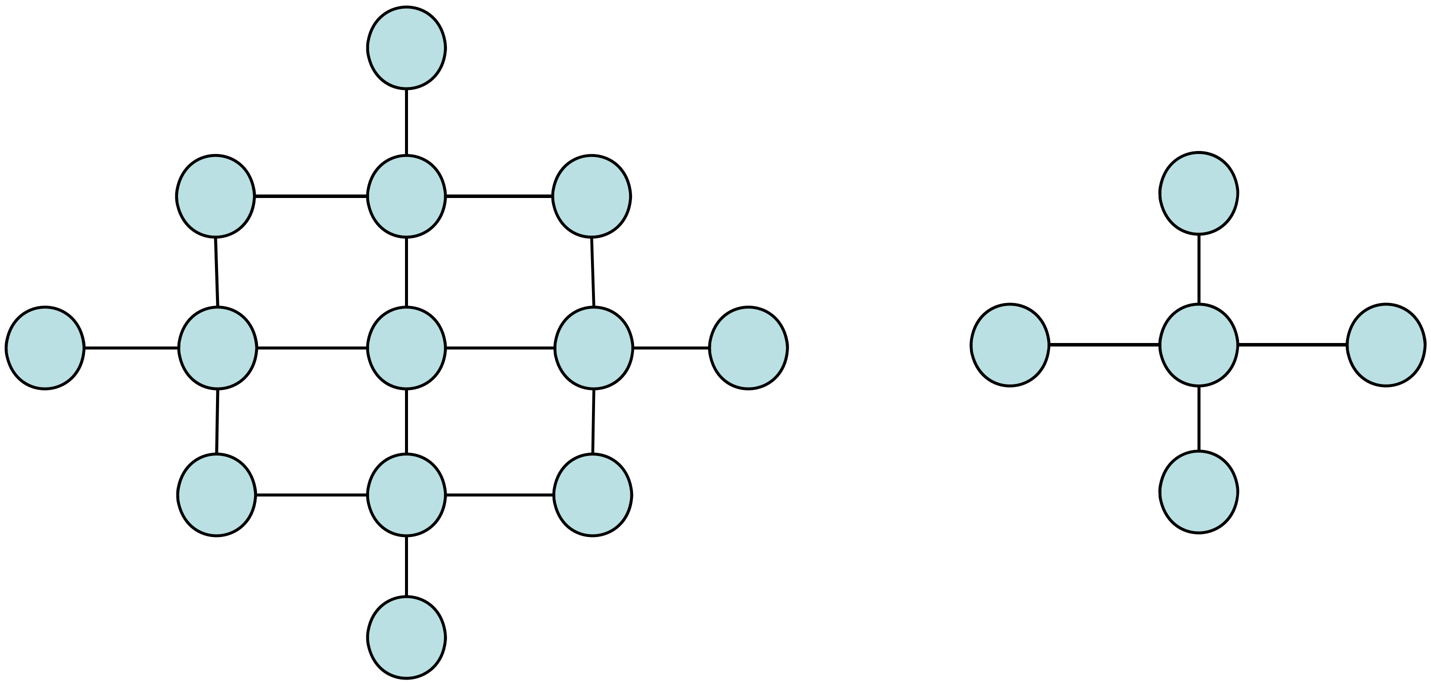
1) visit start, label start as visited.
2) push start to a stack s.
3) while s is not empty
   i) return the top value of s and store it as v
   ii) If v has one unvisited adjacent vertex u, visit u and push u to the stack s.
   iii) If v does not have unvisited adjacent vertices, remove the top value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

The list of vertices visited in order is:

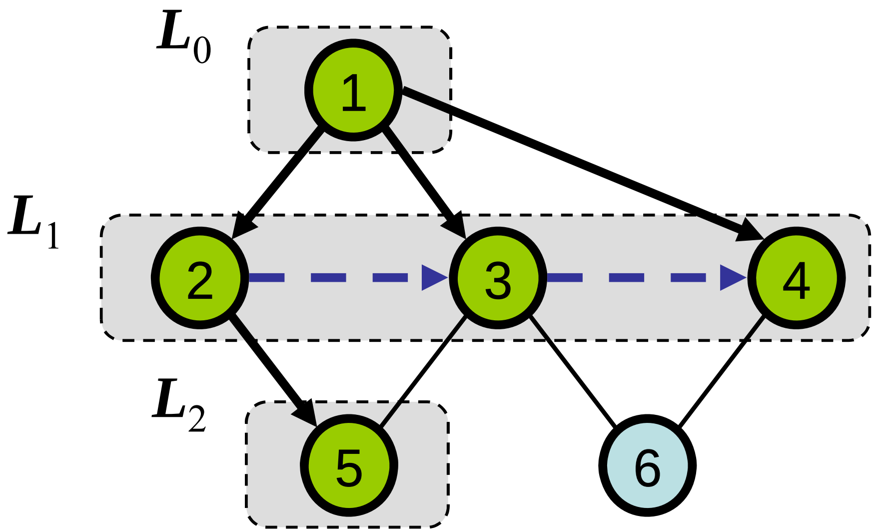
A B D G I H J K L C E F

An Application of DFS

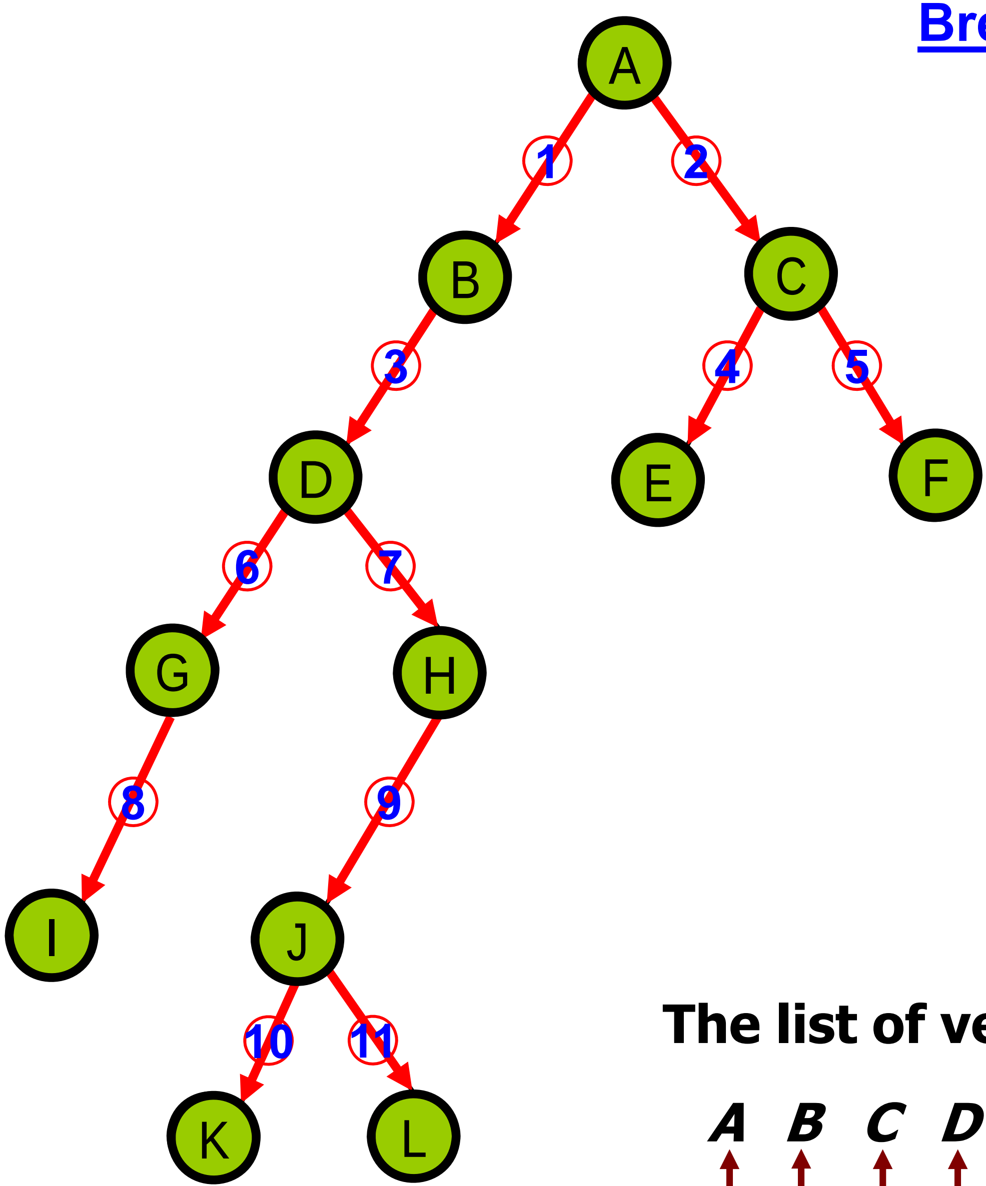
- DFS can be used to test whether a graph is connected.
  - Run DFS using any vertex as the start vertex
  - Upon completing of the algorithm, check whether all vertices are visited
  - The graph is connected if and only if all vertices are visited, i.e. all vertices are reachable from the start vertex



Breadth-First Search



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:

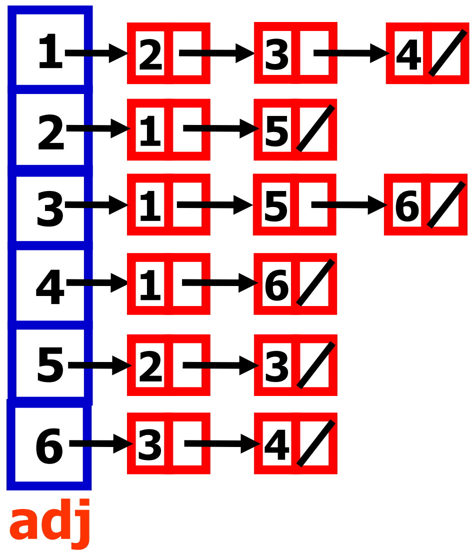
A B C D E F G H I J K L

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()
        u = adj[v]
        while (u != null) {
            if (!visit[u]) {
                visit u
                visit[u] = true
                q.enqueue(u)
            }
            u = u.next
        }
        q.dequeue()
    }
}
```

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



 A/P Huang Guangbin

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.

Breadth-First Search (contd)

The expression

q.enqueue(val)

adds *val* to *q*.

The expression

q.front()

returns the value at the front of *q* but does not remove it.

The expression

q.dequeue()

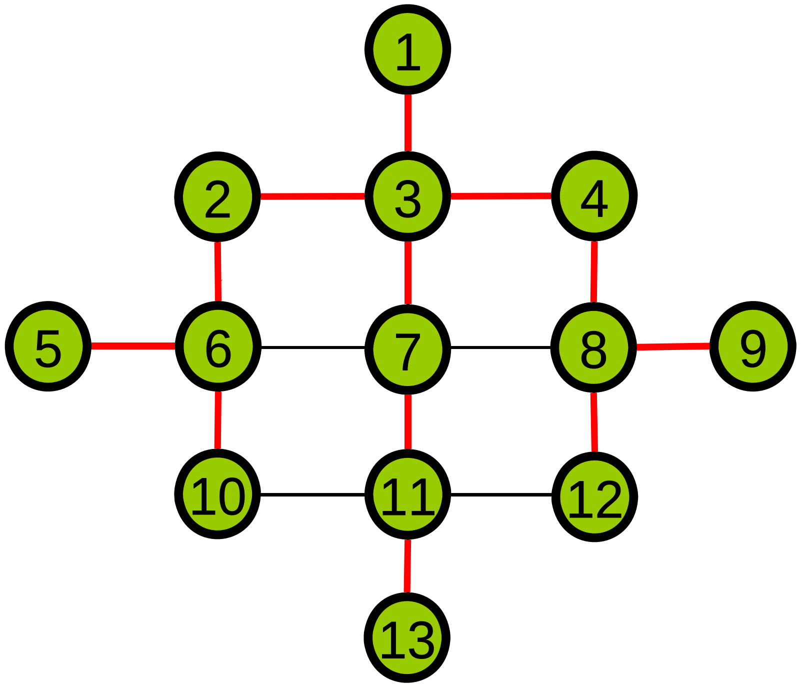
removes the item at the front of *q*.

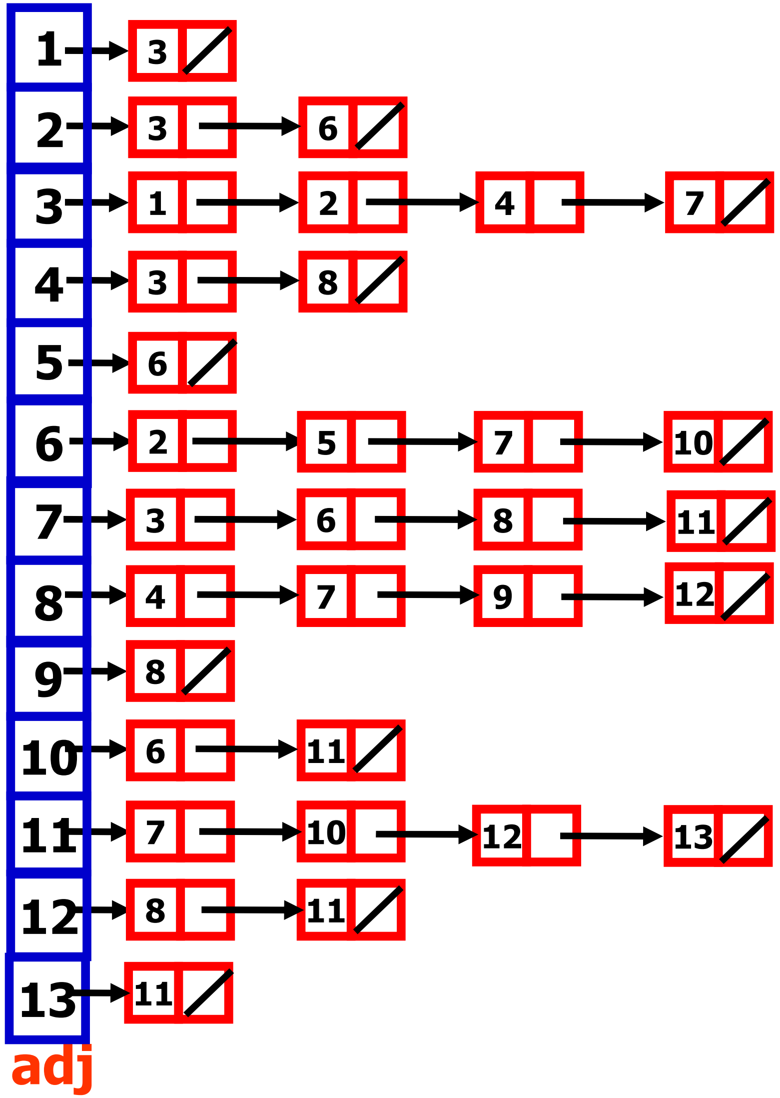
The expression

q.empty()

returns true if *q* is empty or false if *q* is not empty.

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()
        u = adj[v]
        while (u != null) {
            if (!visit[u]) {
                visit u
                visit[u] = true
                q.enqueue(u)
            }
            u = u.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)

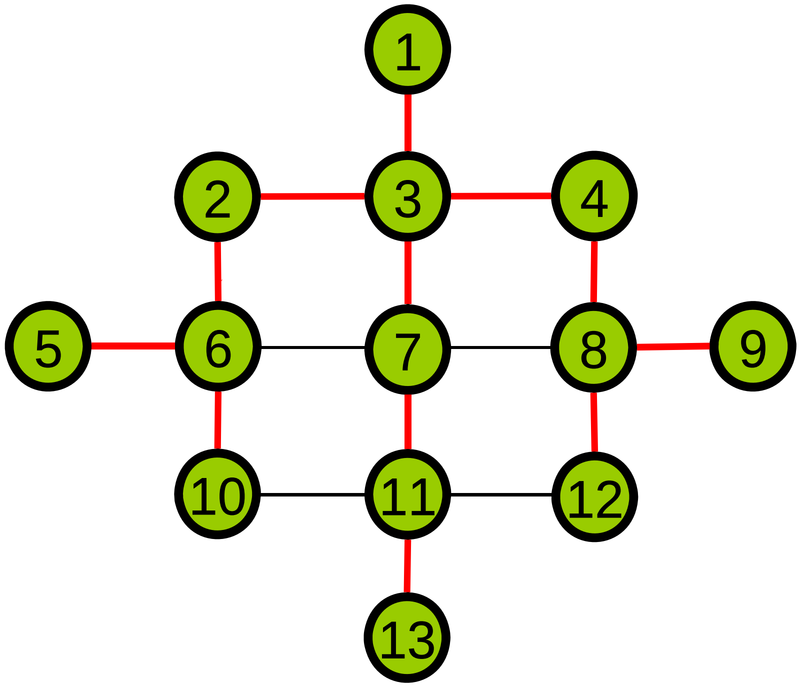
hence time complexity of nested while loops = O(m)

overall time complexity = O(m+n)

Applications

- **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**

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] = ∞
    length[s] = 0
    q.enqueue(s)
    while (!q.empty()) {
        v = q.front()
        u = adj[v]
        while (u != null) {
            if (length[u]==∞) {
                length[u] = 1+ length[v]
                q.enqueue(u)
            }
            u = u.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

Topological Sorting

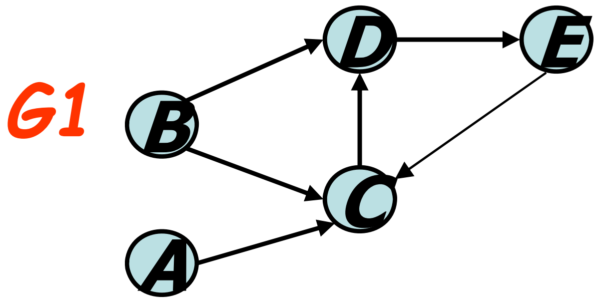
Topological Sort

- In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.

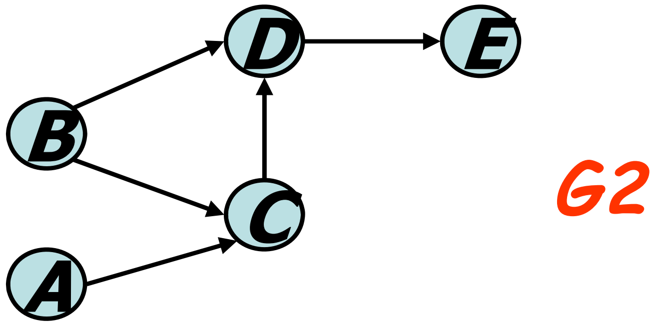
- More formally, define the partial order relation *R* over the nodes of the DAG such that *xRy* if and only if there is a directed path from *x* to *y*. Then, a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.

Topological Sort

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



Is G1 a DAG?



Is G2 a DAG?

Let v be a vertex in a digraph.

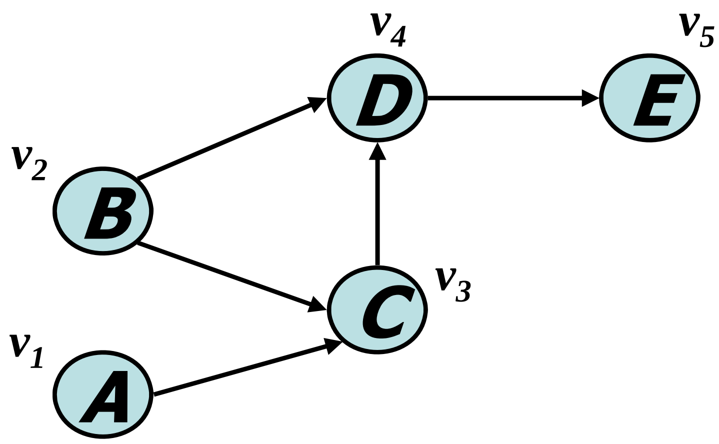
in-degree of v = number of incoming edges;

out-degree of v = number of outgoing edges

Eg: in-degree of vertex C in G1 = 3

out-degree of vertex C in G1 = 1

Topological Sort



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

- **A topological sorting of a DAG is a numbering**

  $v_1, \ldots, v_n$

  of the vertices such that for every edge (v_i, v_j)**,**

  we have $v_i < v_j$

Topological Sort

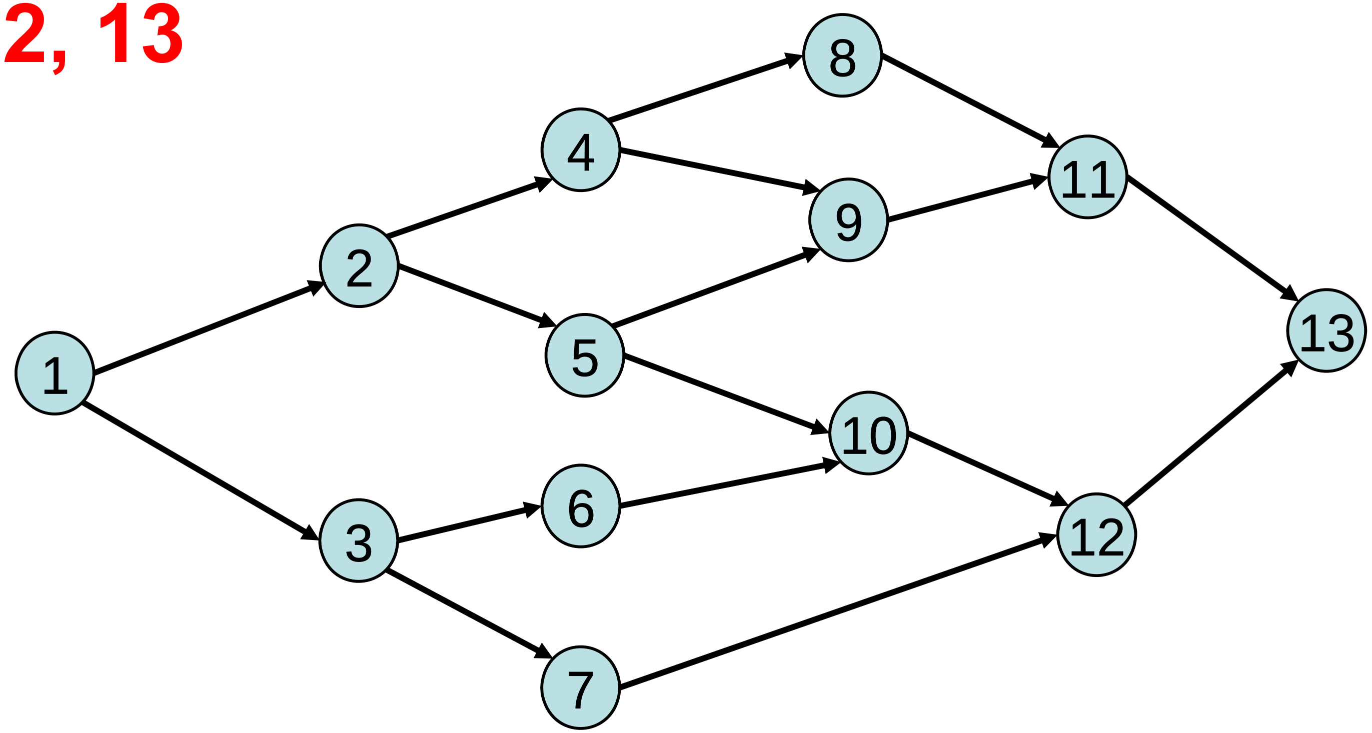
- **We will discuss:**
  - **the idea of sorting elements in a DAG**
    - **examples of topological sort (**pp. 78-120, from D. W. Harder, University of Waterloo)
  - **the implementation**
    - **using a table of in-degrees**
    - **using DFS**

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
- Thus, it must be possible to list all of the vertices such that in that list, v_i preceeds 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

- Such an ordering is called a ***topological sort***
- For example, in this DAG, one topological sort is **1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 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

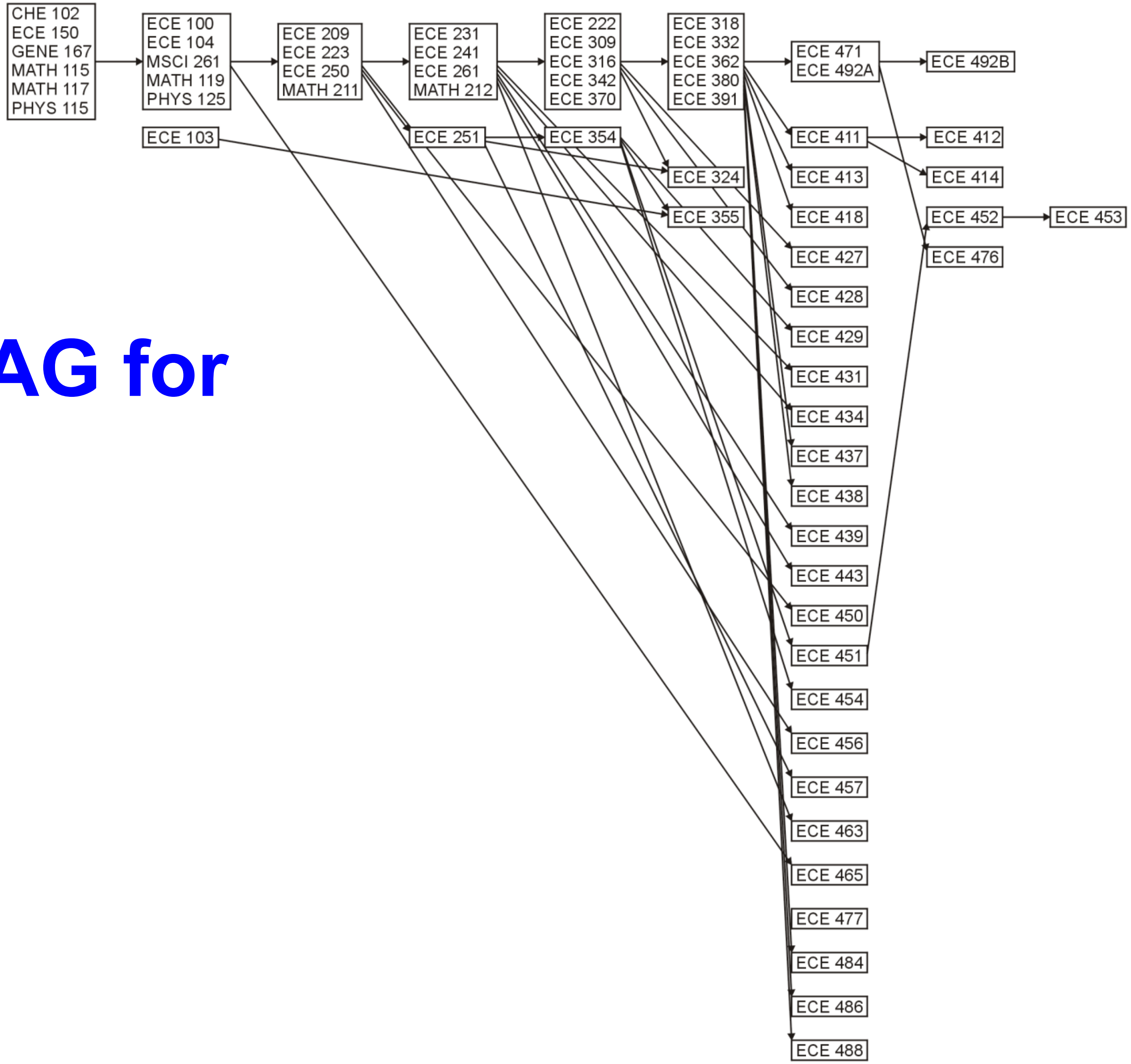
A/P Huang Guangbin

Application

- Another set of *tasks* are courses
- Certain courses have prerequisites
- The resulting graph is a DAG
- Interesting twist: corequisites (courses which must be taken concurrently)
  - ☞treat them as a single node

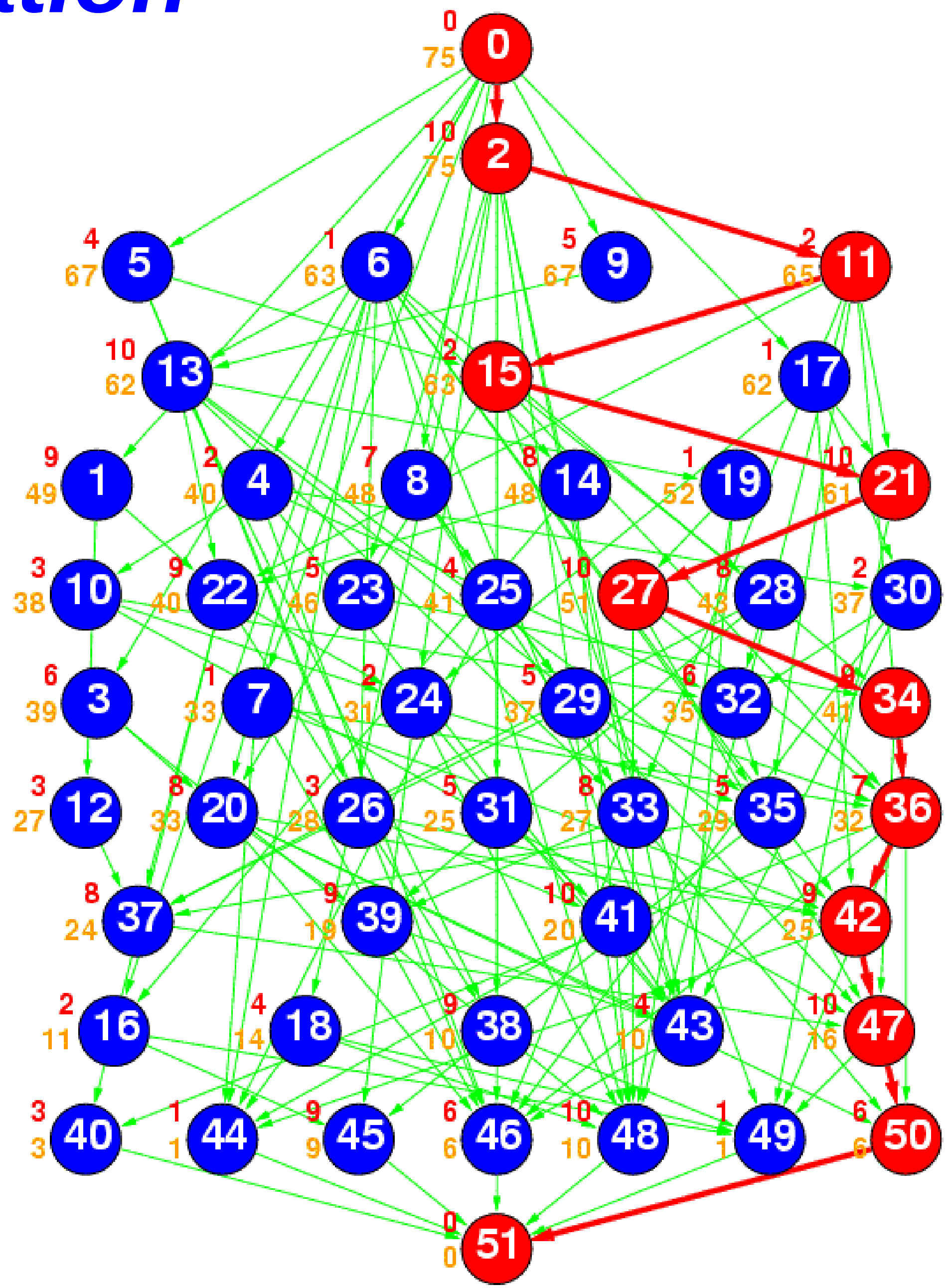
Applications

- The course sequence DAG for ECE students



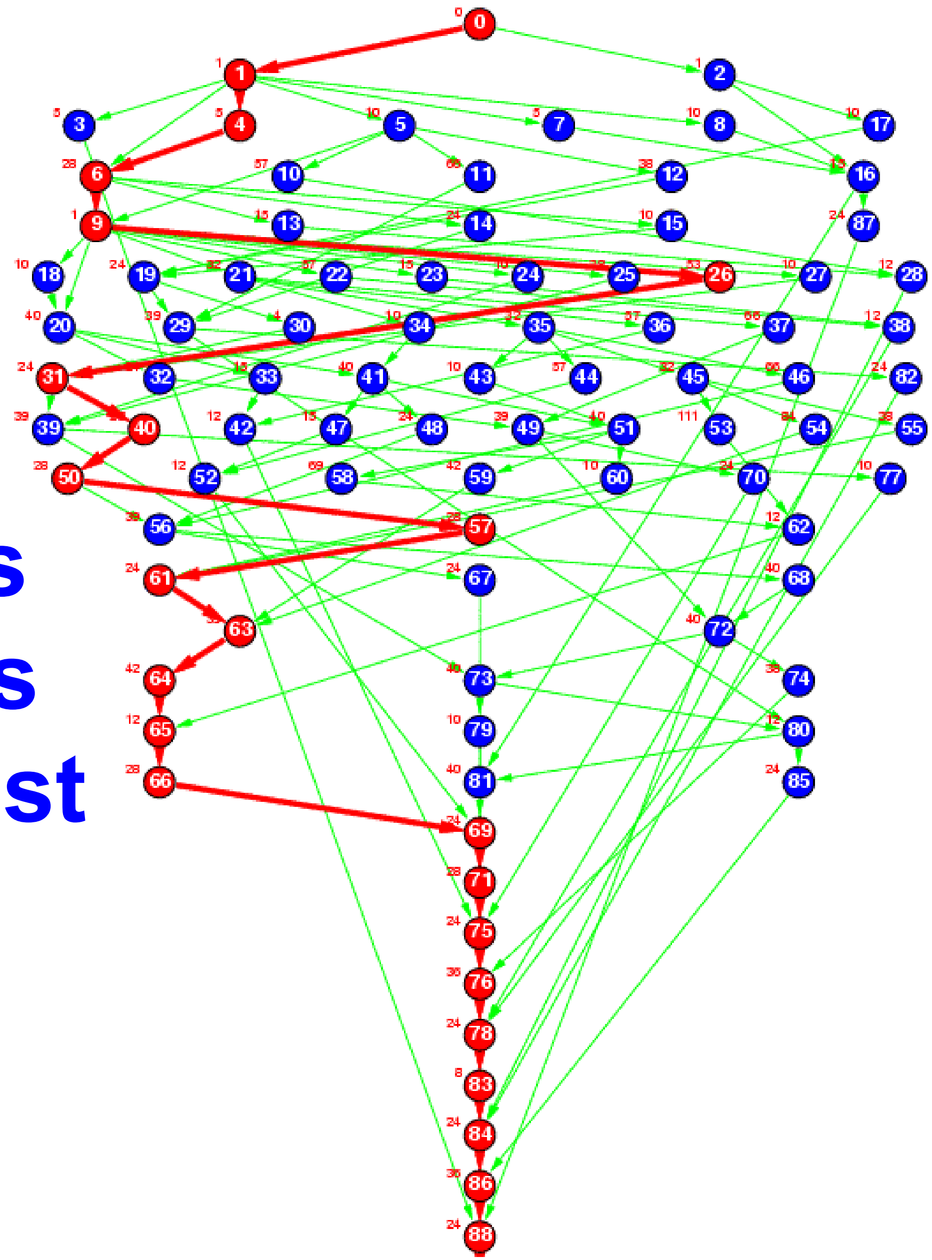
Application

- The following is a DAG representing a number of tasks
- The numbering indicates the topological sort of the tasks
- The green arrows represent *precedence constraints*



Application

- **Here we see another topological sort of a task DAG**
- **The red *critical path* is that sequence of tasks which takes the longest time**

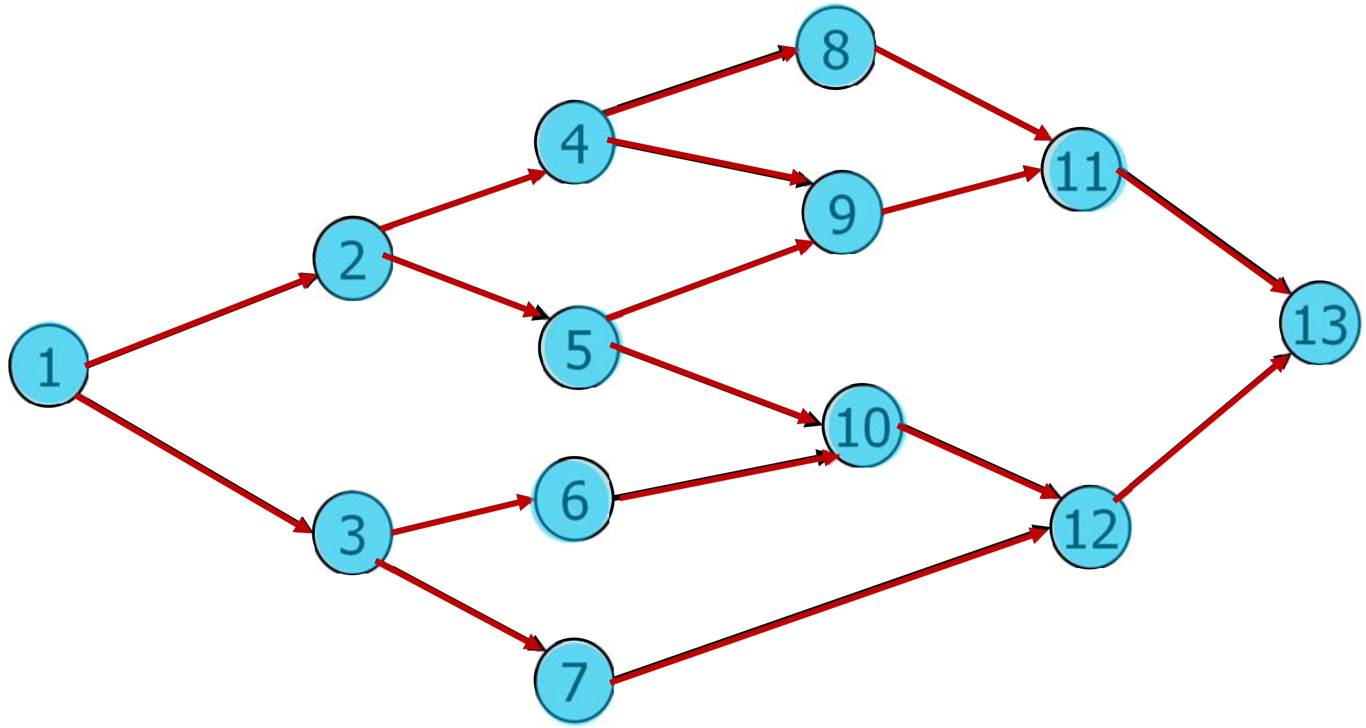


Application

- This application would be used for performing these tasks on m processors
- In this case, the topological sort takes into case other considerations, specifically, minimizing the total run time of the collection of tasks

Topological Sort

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



The list of vertices visited in order is:

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

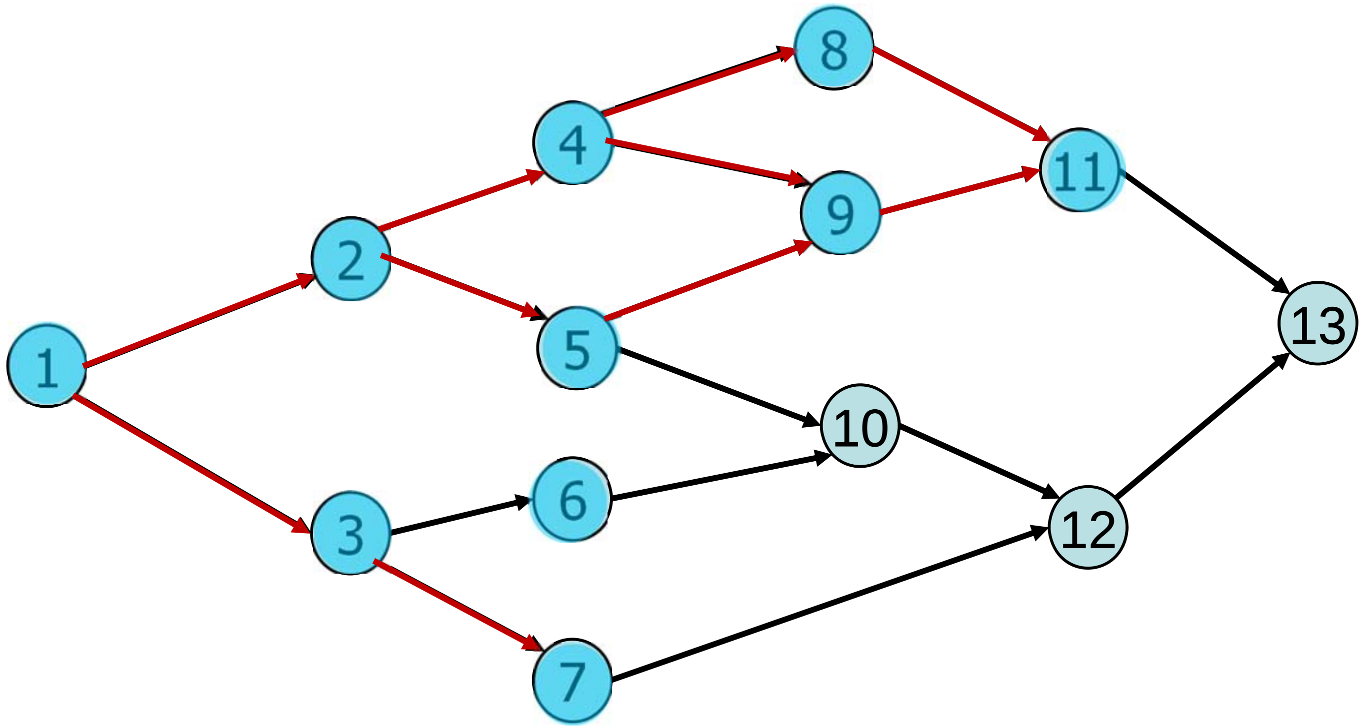
Topological Sort

- 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:**

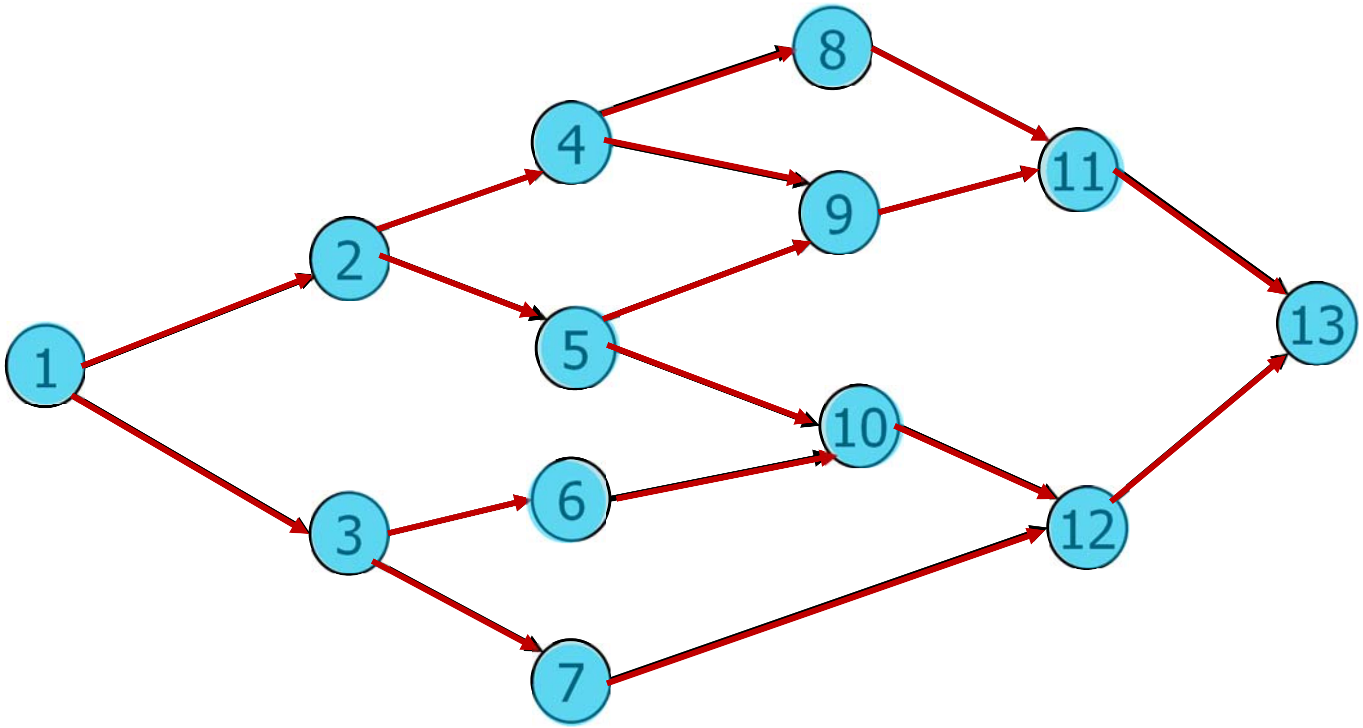
1, 2, 4, 8, 5, 9, 11, 3, 7



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

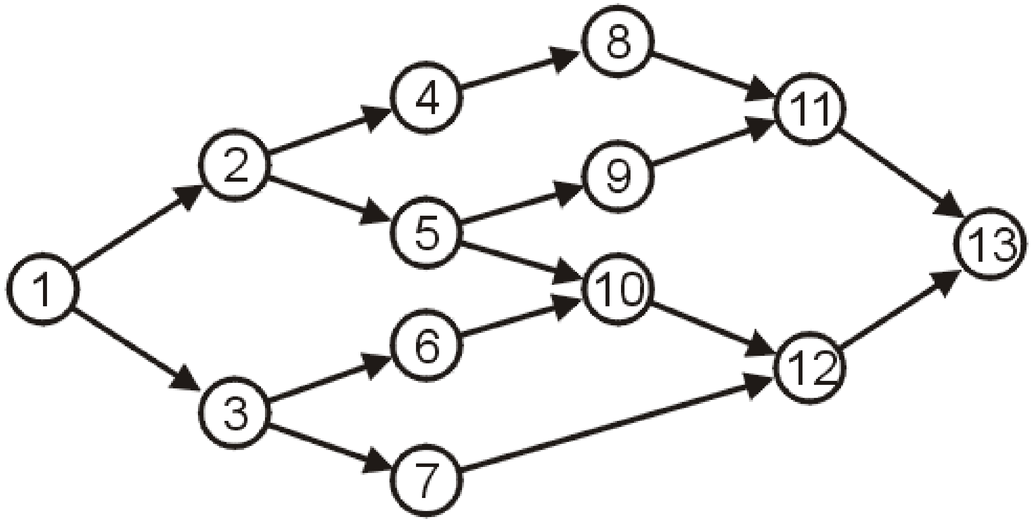
- As seen before, these are not the only topological sorts possible for this graph, for example

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

  is equally acceptable

Implementation

- What are the tools necessary for a topological sort?
- Suppose first we keep an array of the in-degree of each of the vertices:



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

Implementation

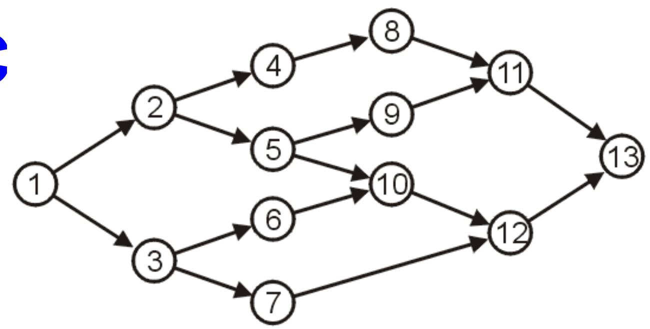
- Now, think back to the breadth-first traversal
  - there, we placed the root vertex into a queue
- In this case, create some sort of container (stack or queue) which initially contains all vertices with an in-degree of 0

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

Implementation

- We will initially use the terminology of a queue
- Initially, this queue only contains vertex 1
- We can then dequeue the head of the queue (in this case 1) and add this to our topologic

1

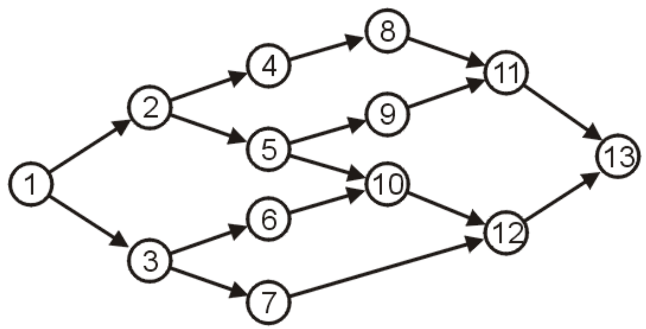


1	0
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	2
11	2
12	2
13	2

Implementation

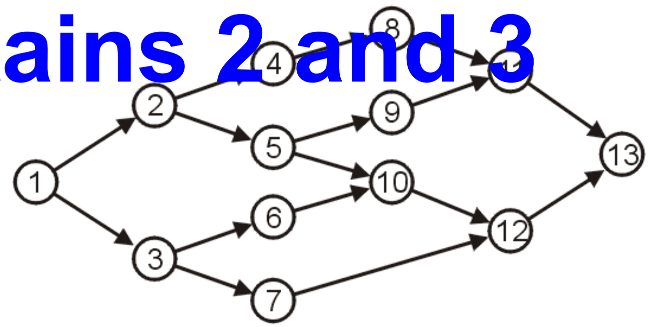
- With a breadth-first traversal, we enqueued all of the dequeued vertices children
- In this case, we will decrement the in-degree of all vertices which are adjacent to the dequeued vertex

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



Implementation

- Each time we decrement the in-degree of a vertex, we check if the in-degree is reduced to zero
- If the in-degree is reduced to zero, we enqueue that vertex
- Consequently, our queue now contains 2 and 3



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

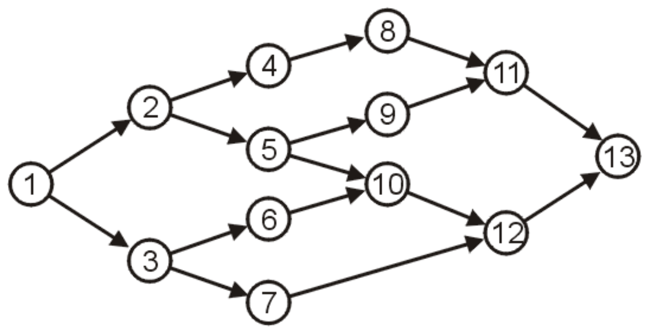
Implementation

- Next, we dequeue 2, add it to our topological sort

  1, 2

  and decrement the in-degrees of vertices 4 and 5
- Both 4 and 5 are reduced to zero, and thus our queue contains

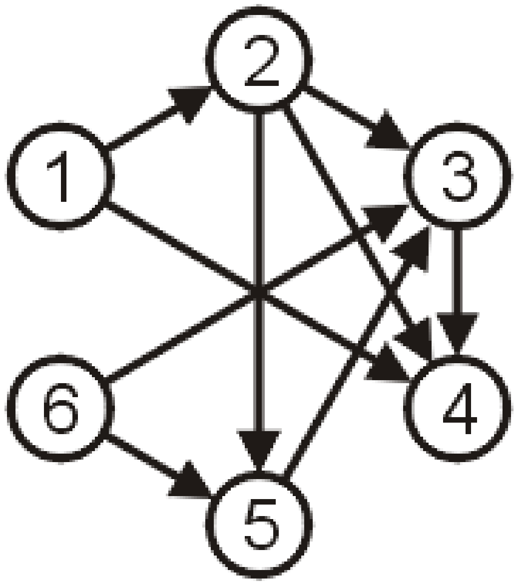
  3, 4, 5



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

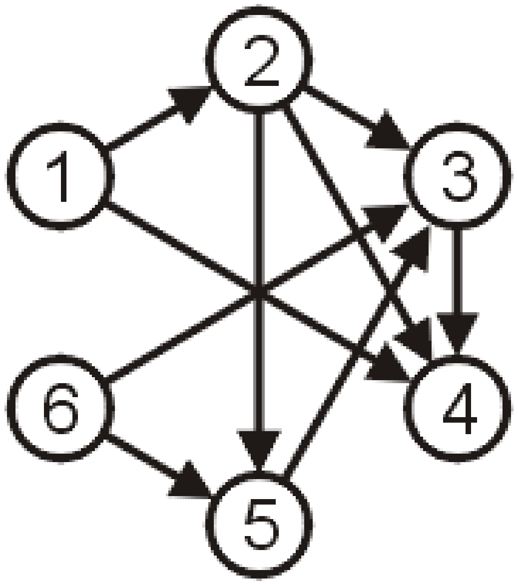
Example

- Consider the following DAG with six vertices



Example

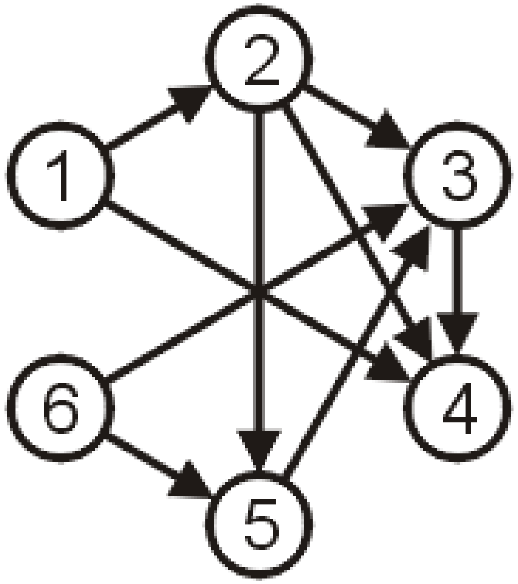
- Let us define the array of in-degrees



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

Example

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



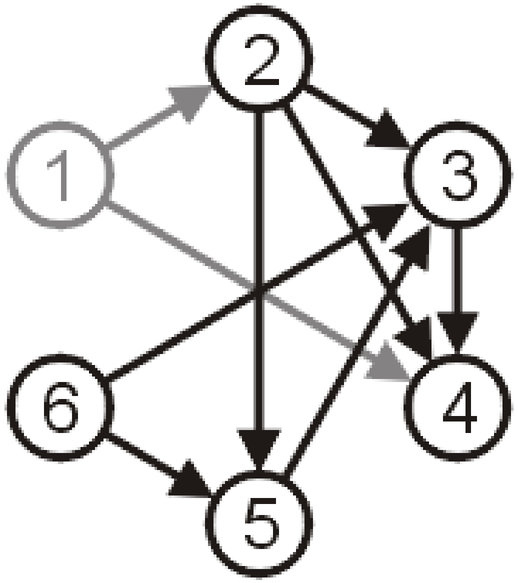
1	0
2	1
3	3
4	3
5	2
6	0

Queue

1	6		

Example

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



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

Queue

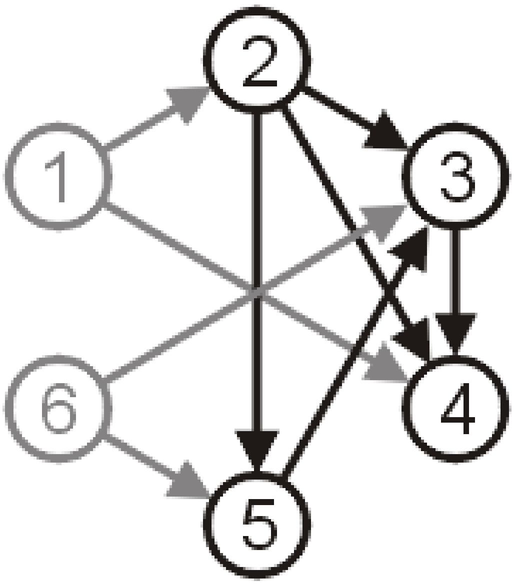
6	2		

Sort

1

Example

- **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

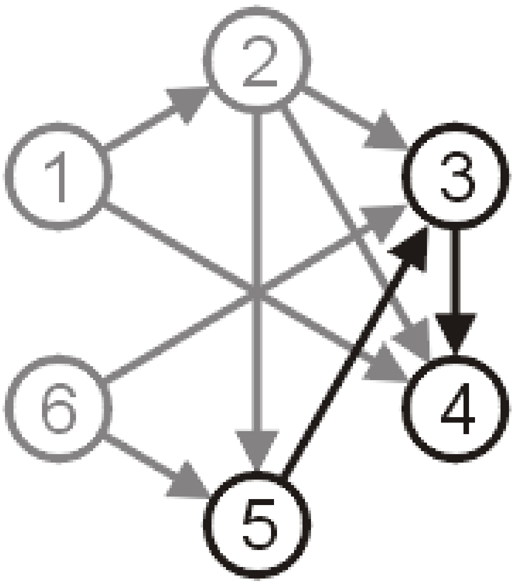
2			

Sort

1, 6

Example

- We dequeue 2, decrement, and enqueue vertex 5



Queue

5			

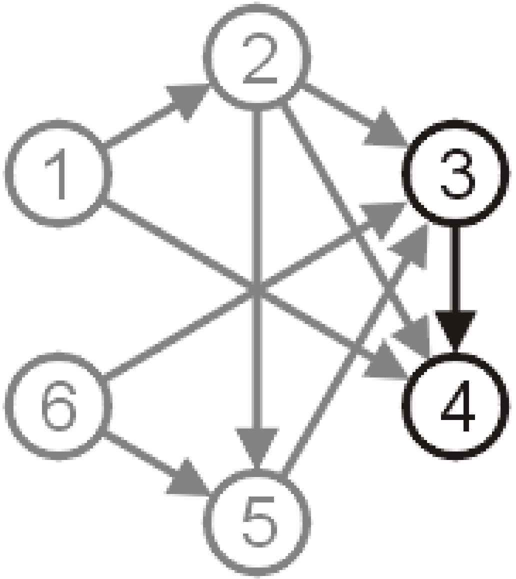
Sort

1, 6, 2

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

Example

- We dequeue 5, decrement, and enqueue vertex 3



Queue

3			

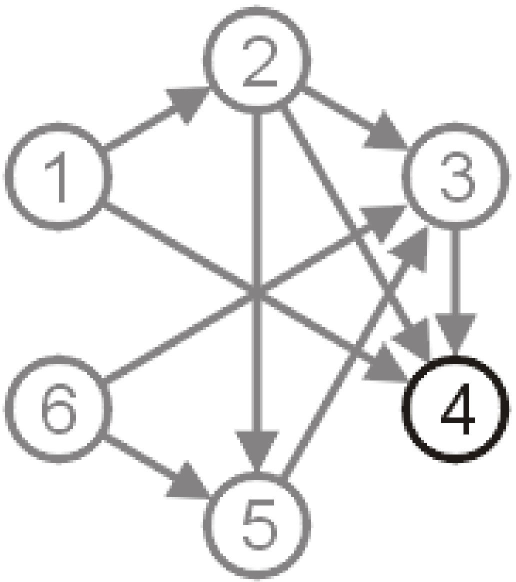
Sort

1, 6, 2, 5

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

Example

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



Queue

4			

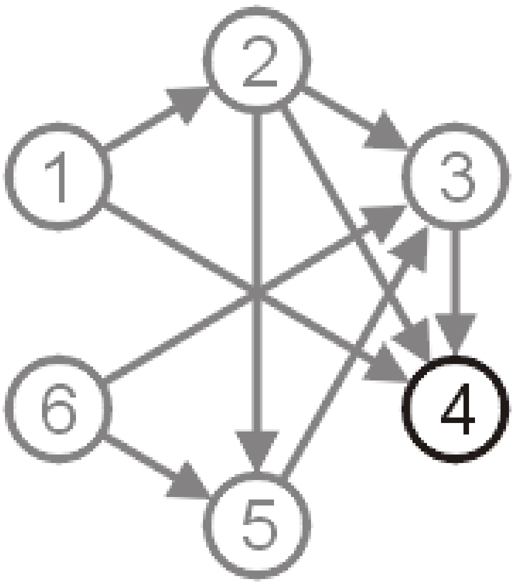
Sort

1, 6, 2, 5, 3

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

Example

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



Queue

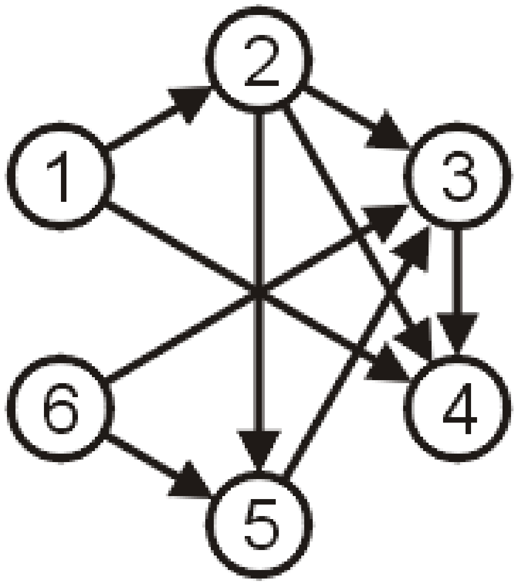
Sort

1, 6, 2, 5, 3, 4

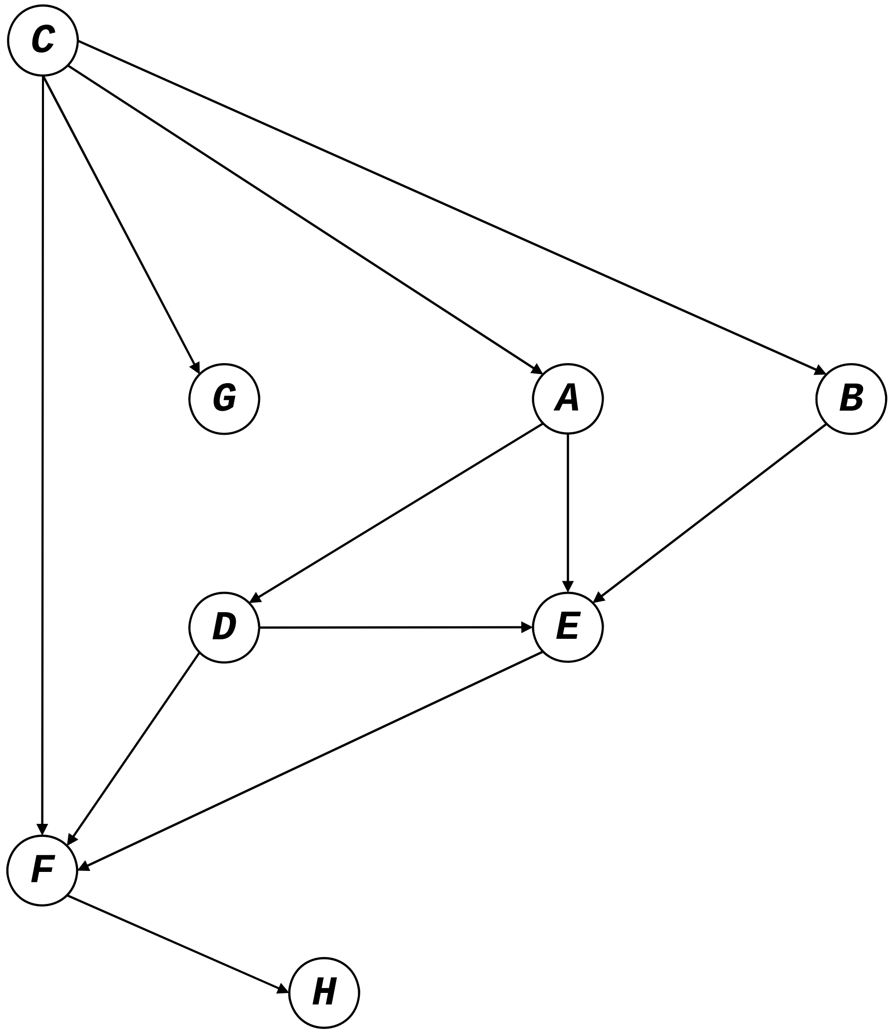
1	0
2	0
3	0
4	0
5	0
6	0

Example

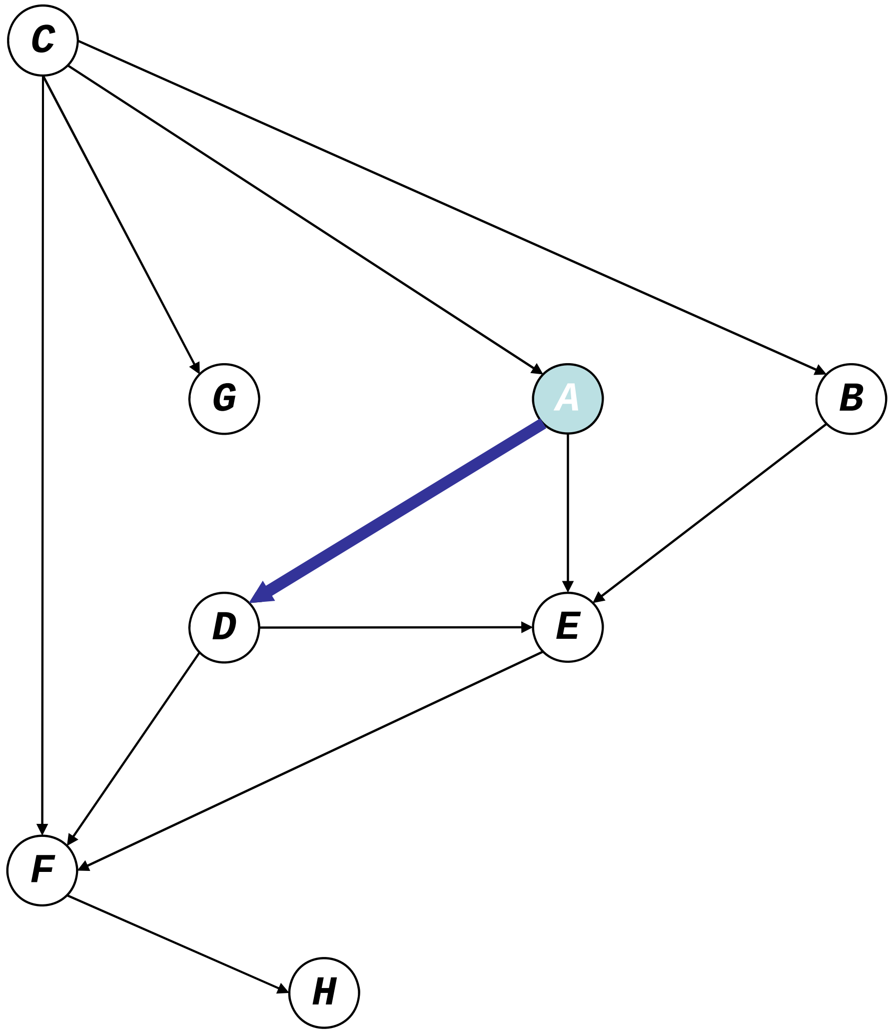
- The queue is now empty, so a topological sort is 1, 6, 2, 5, 3, 4



Topological Sort: DFS



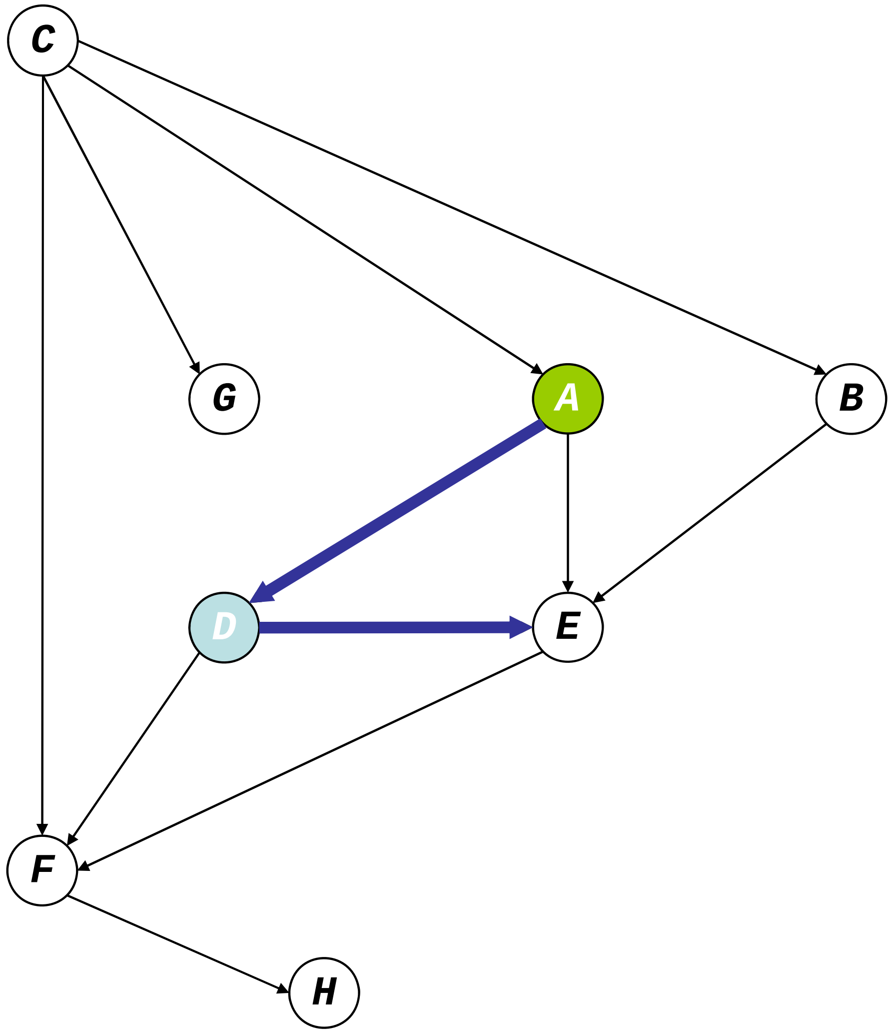
Topological Sort: DFS



`dfs(A)`

A/P Huang Guangbin

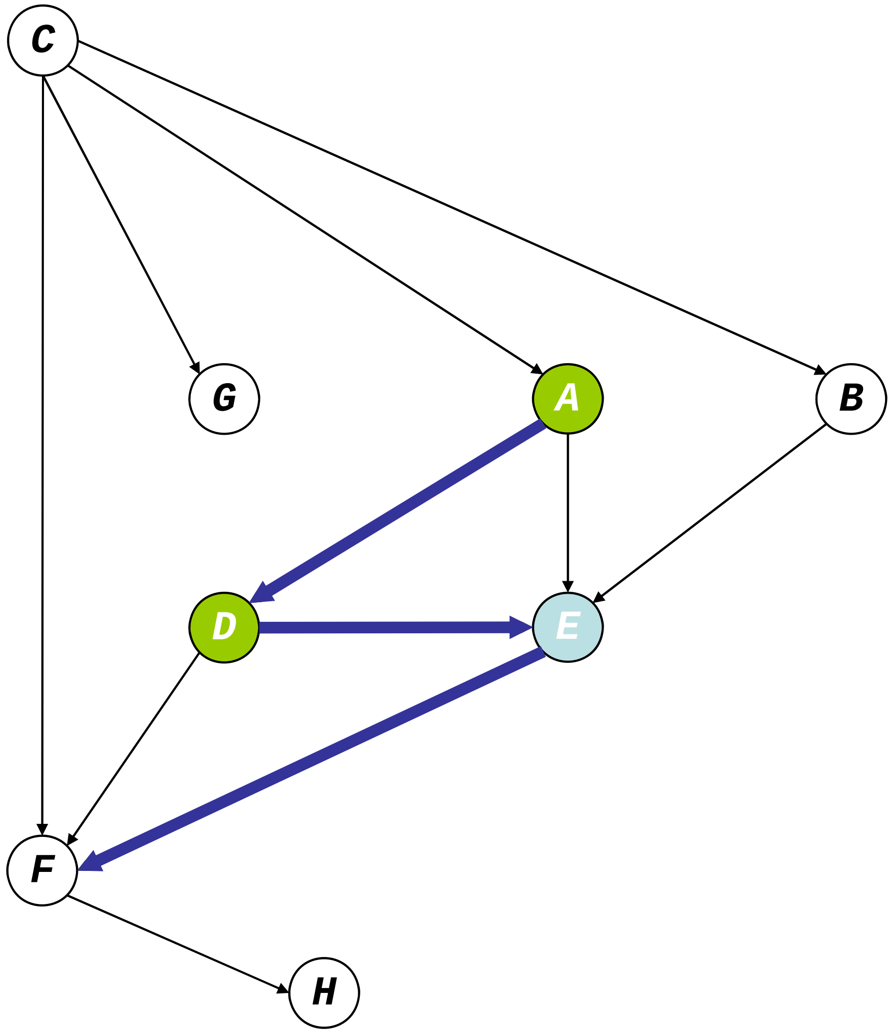
Topological Sort: DFS




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

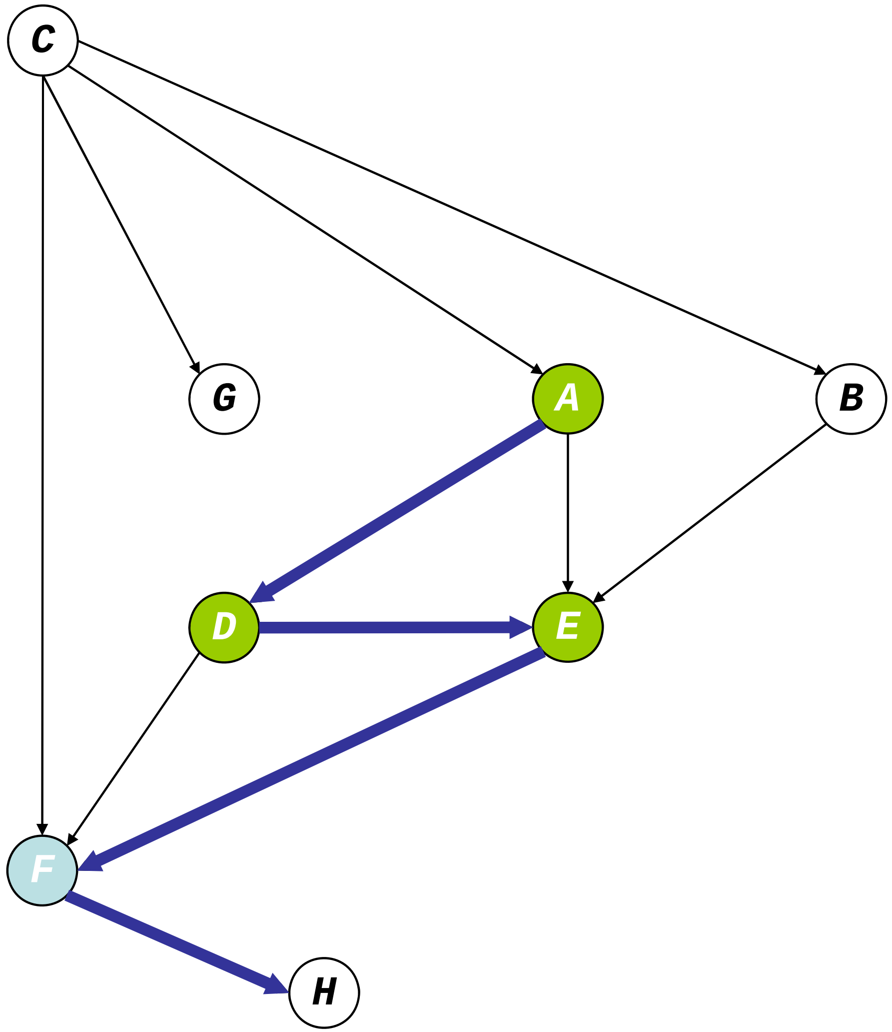

A/P Huang Guangbin

Topological Sort: DFS



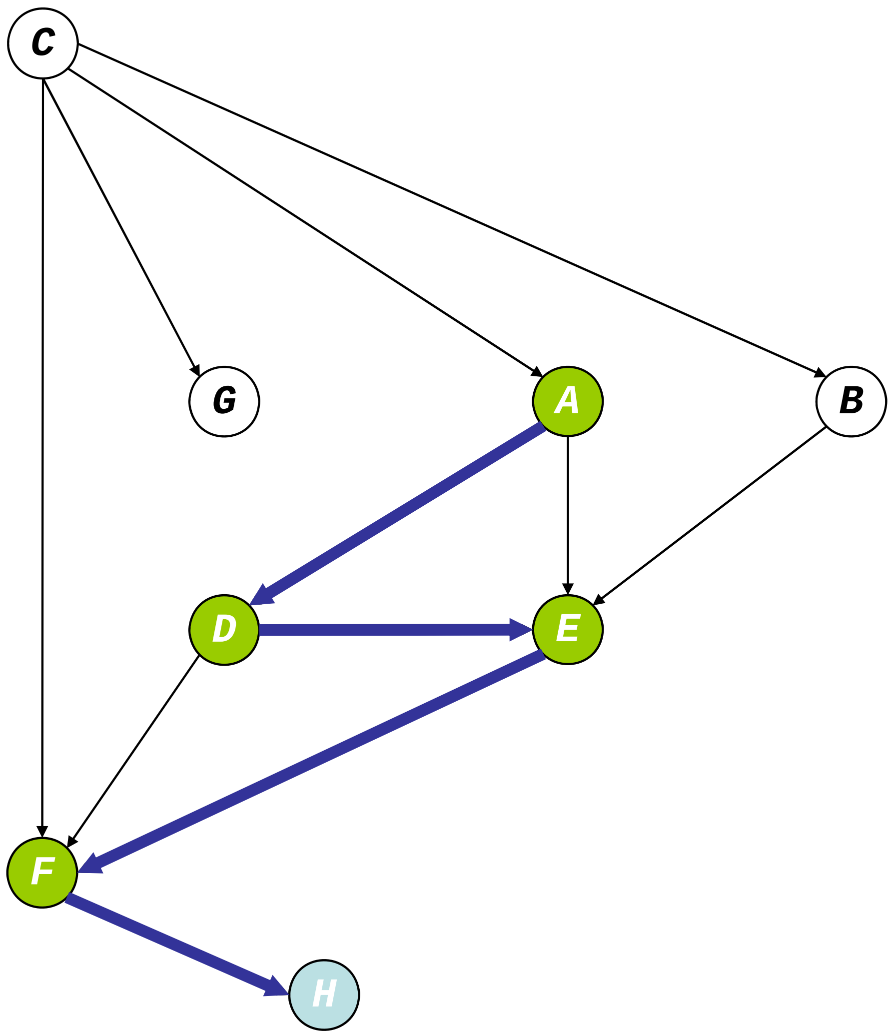

```
dfs(A)
  dfs(D)
    dfs(E)
```

Topological Sort: DFS




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

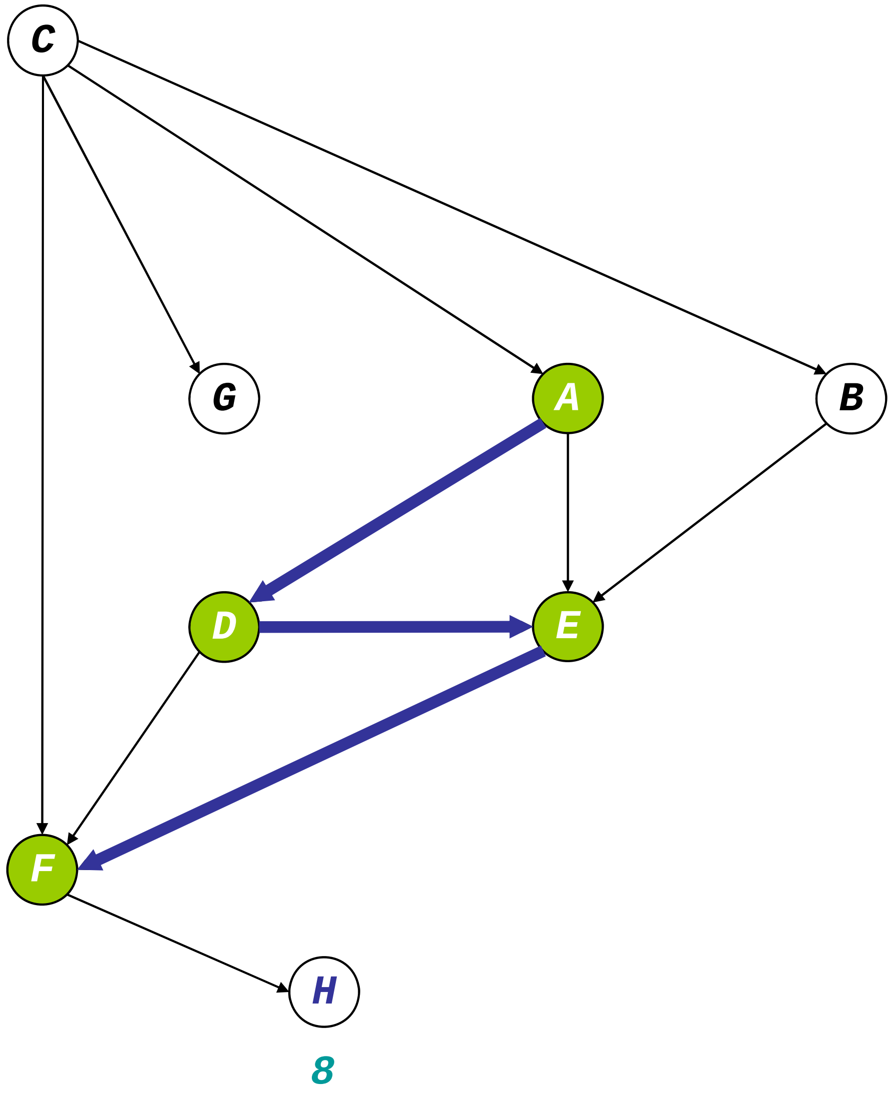
Topological Sort: DFS




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

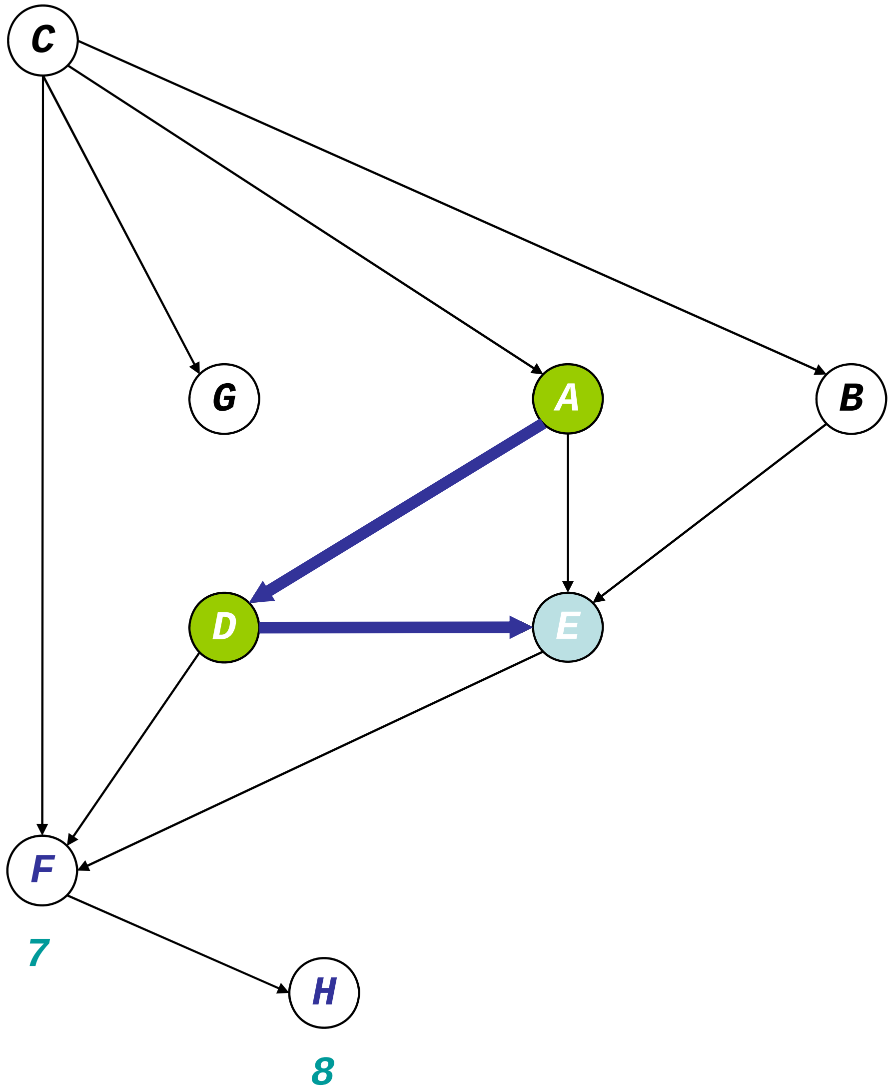

A/P Huang Guangbin

Topological Sort: DFS




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

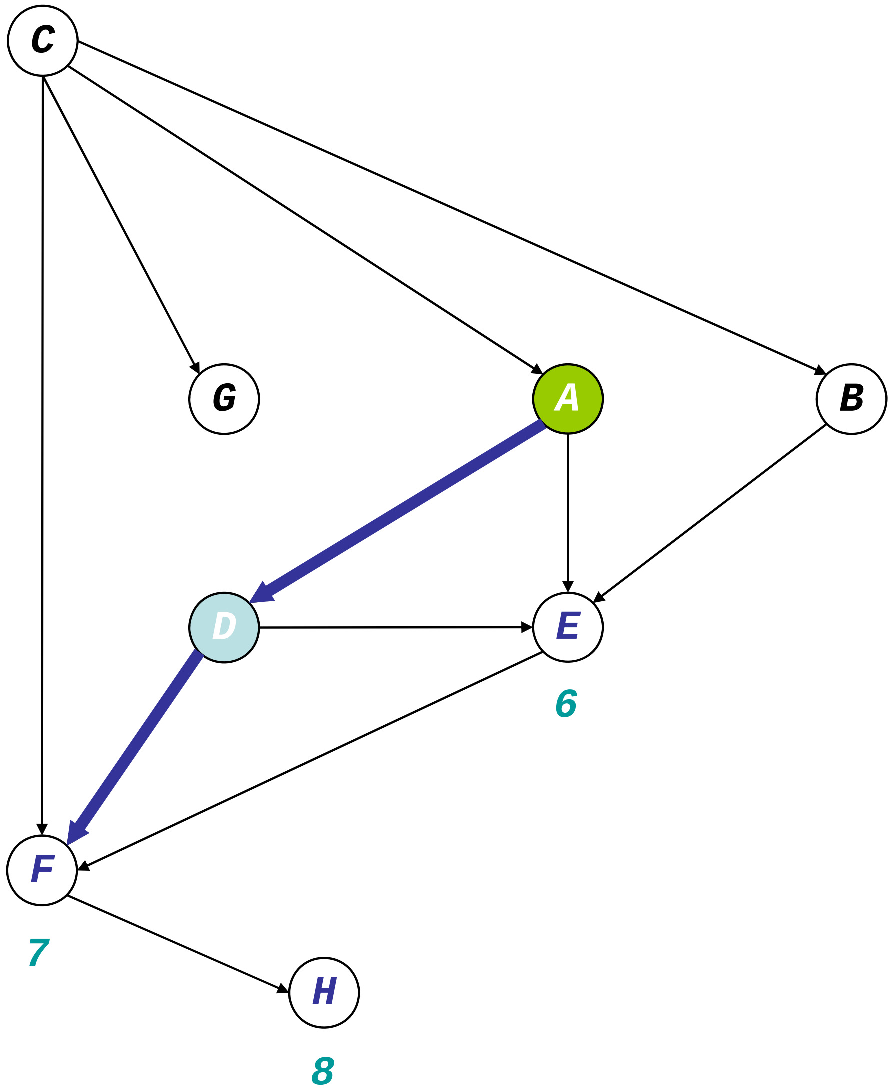
Topological Sort: DFS




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


A/P Huang Guangbin

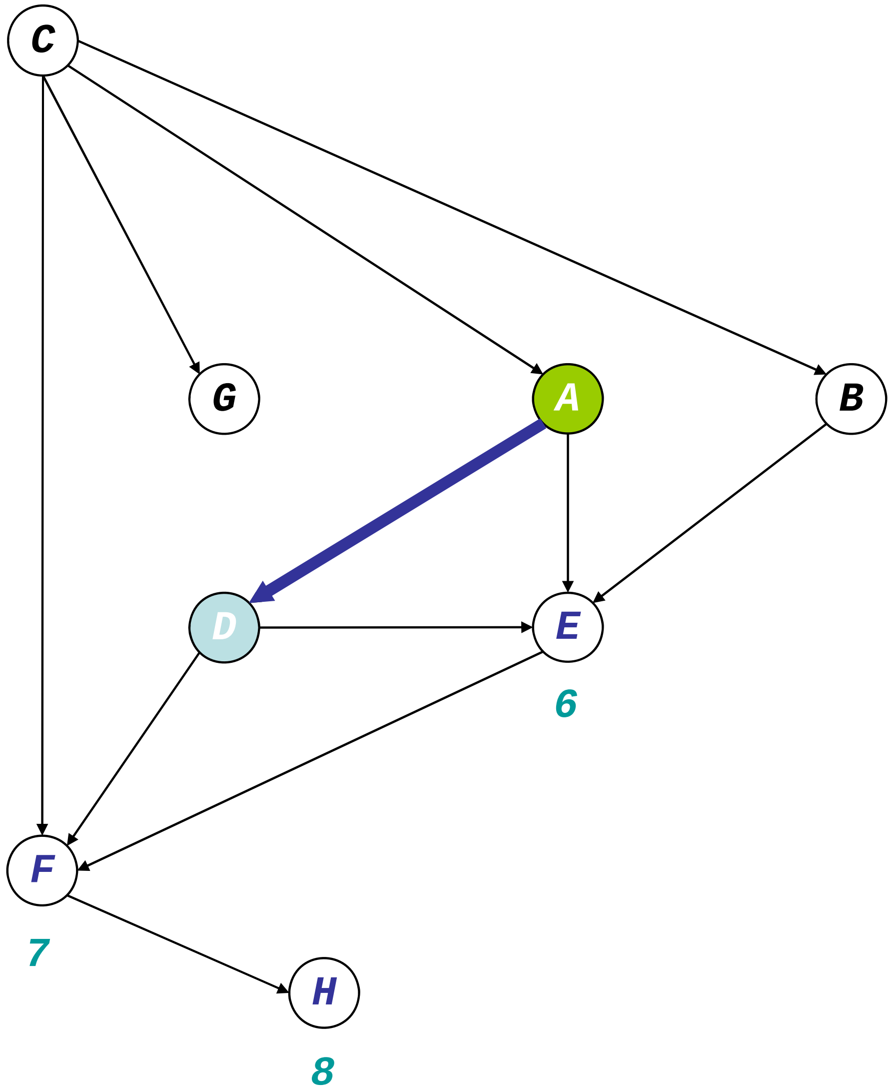
Topological Sort: DFS




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

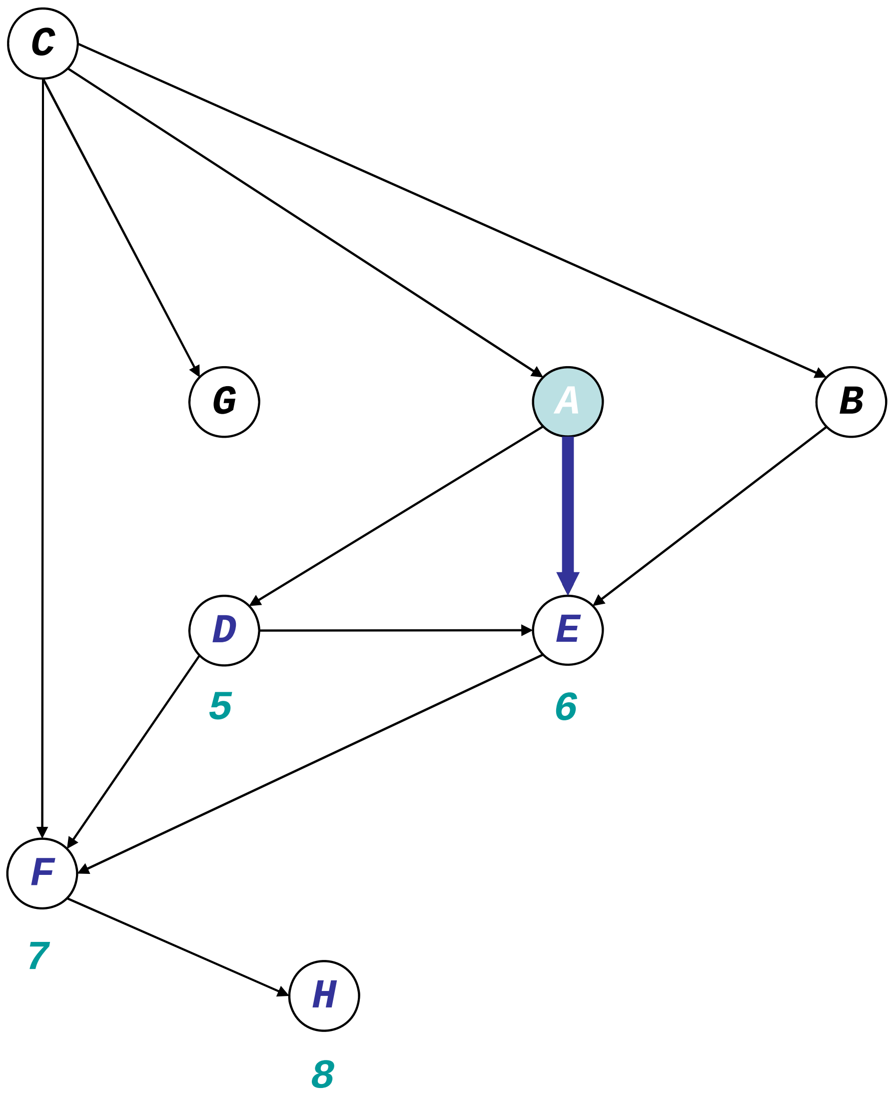

A/P Huang Guangbin

Topological Sort: DFS



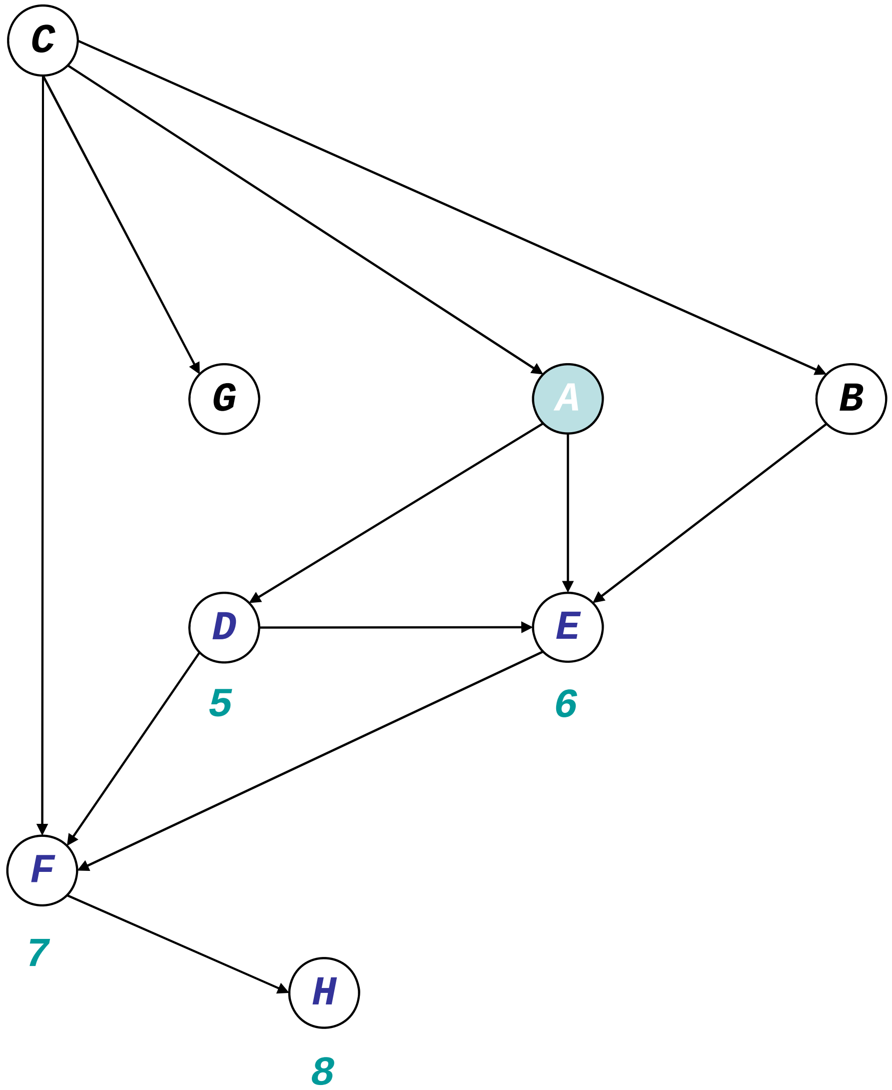

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS



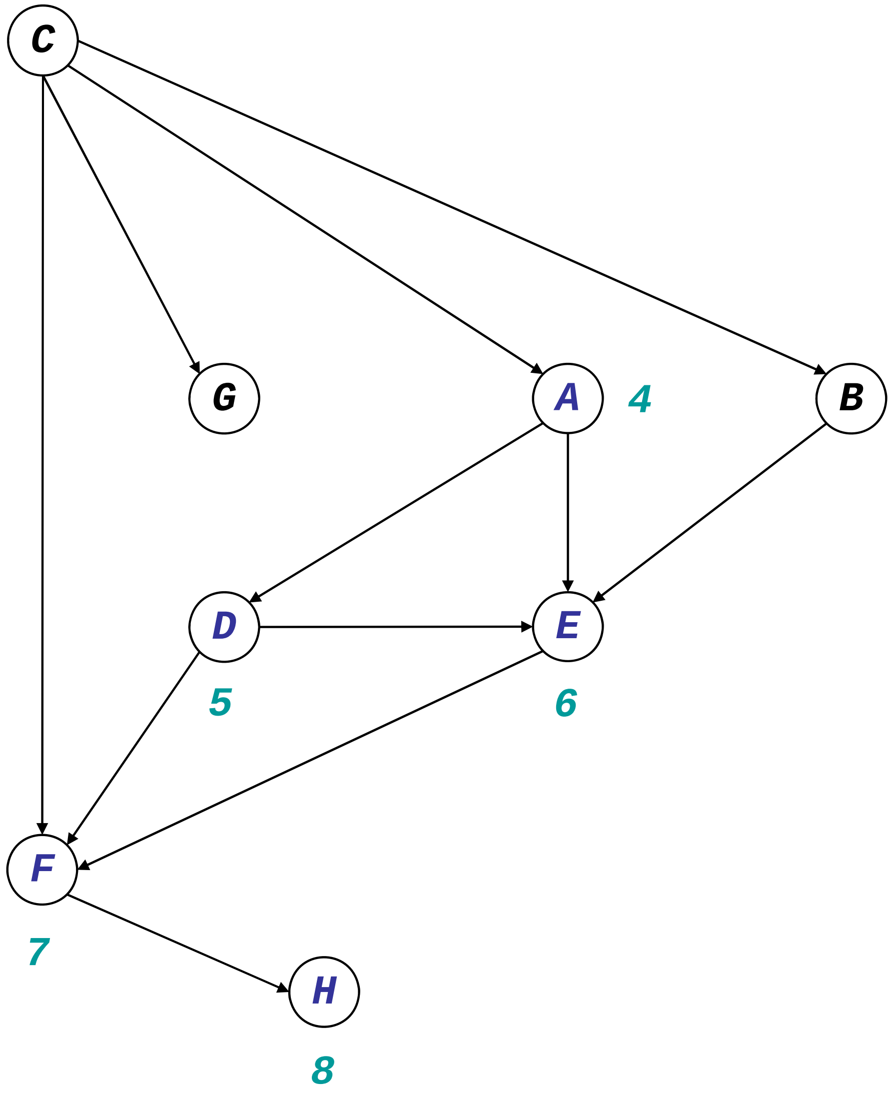
`dfs(A)`

Topological Sort: DFS

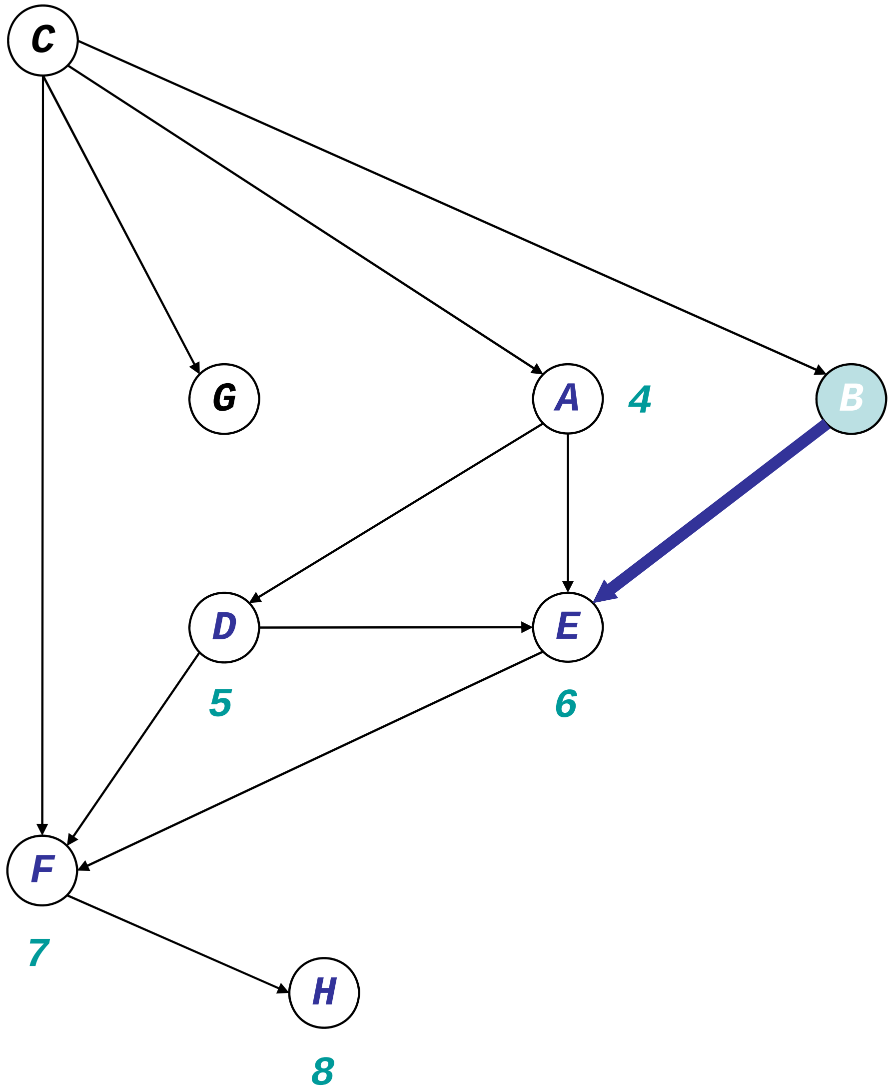


`dfs(A)`

Topological Sort: DFS

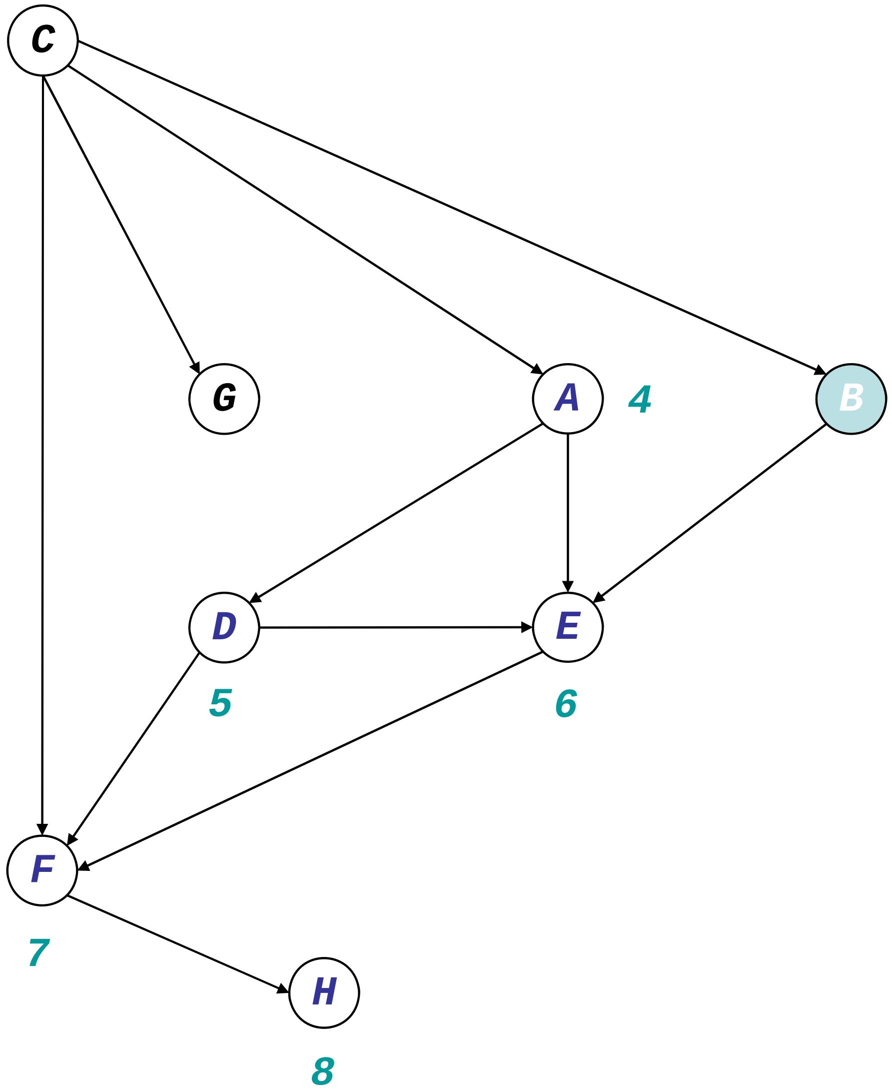


Topological Sort: DFS



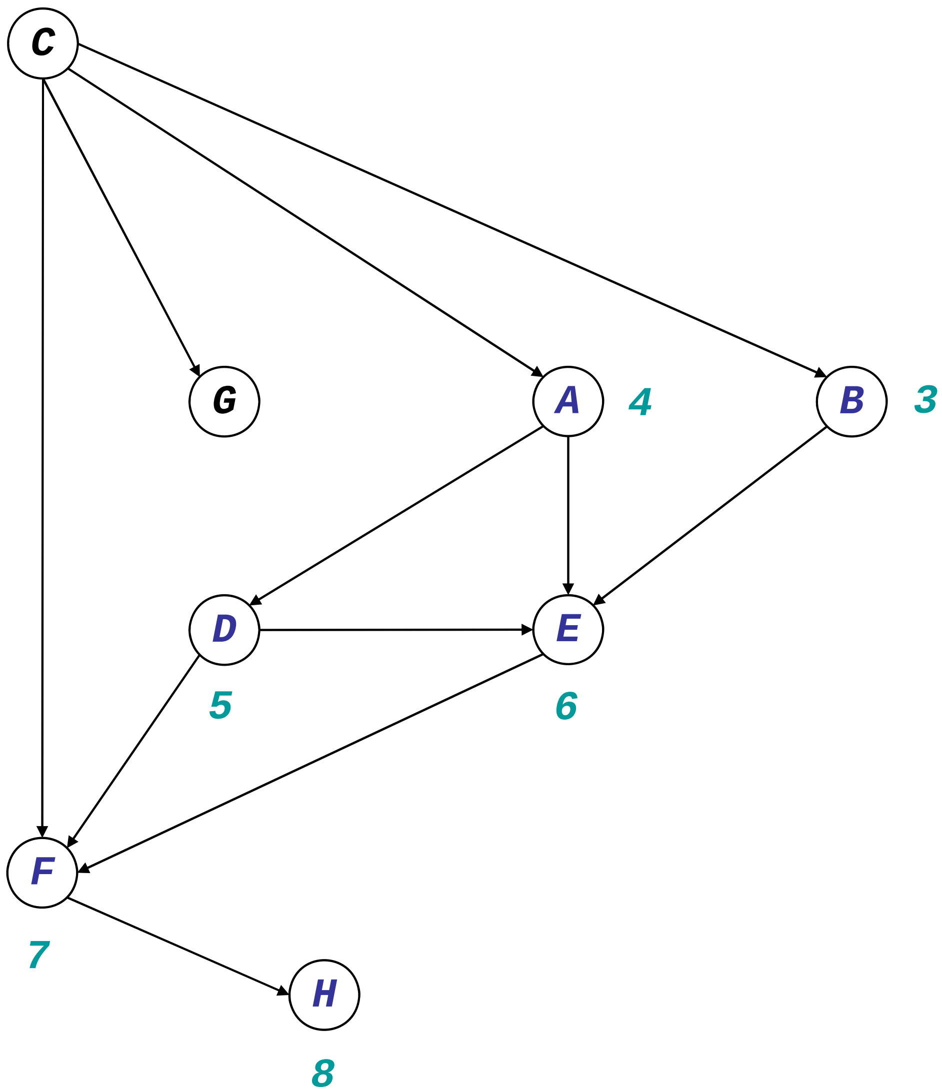
`dfs(B)`

Topological Sort: DFS

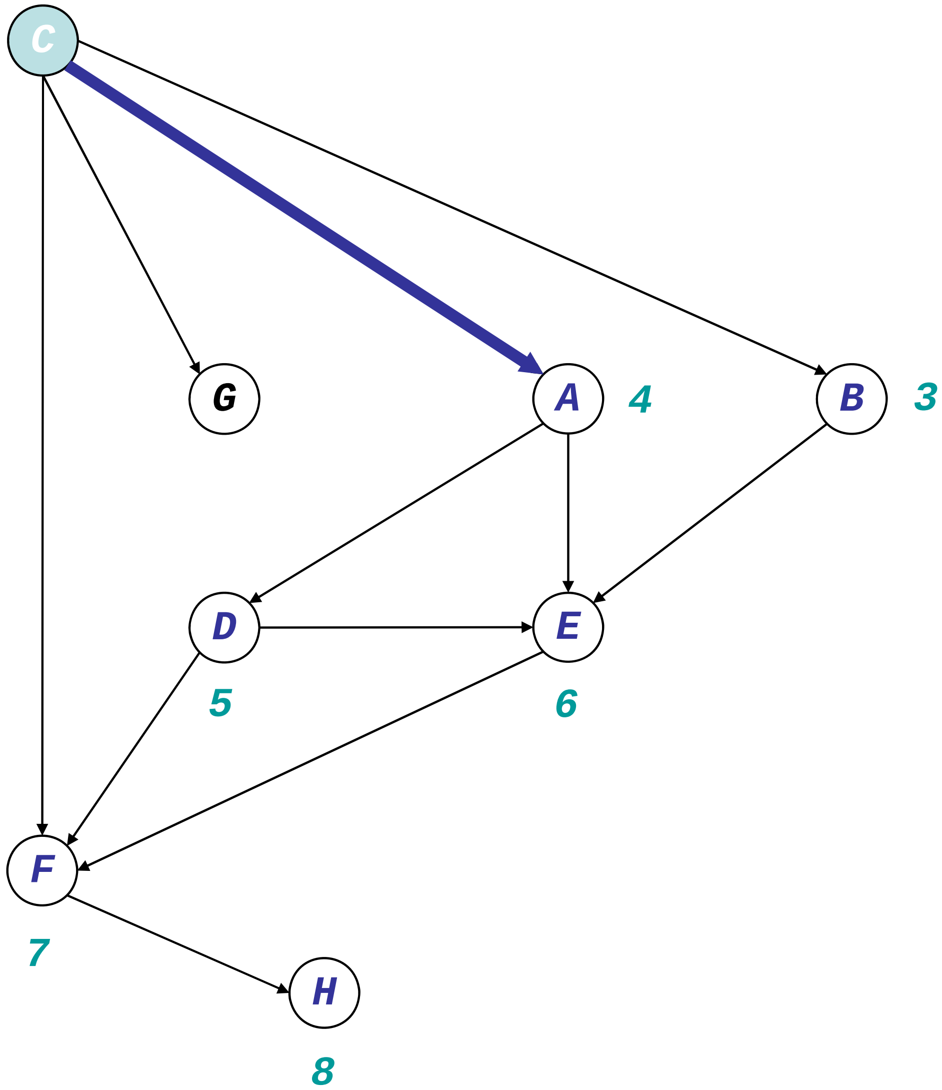


`dfs(B)`

Topological Sort: DFS

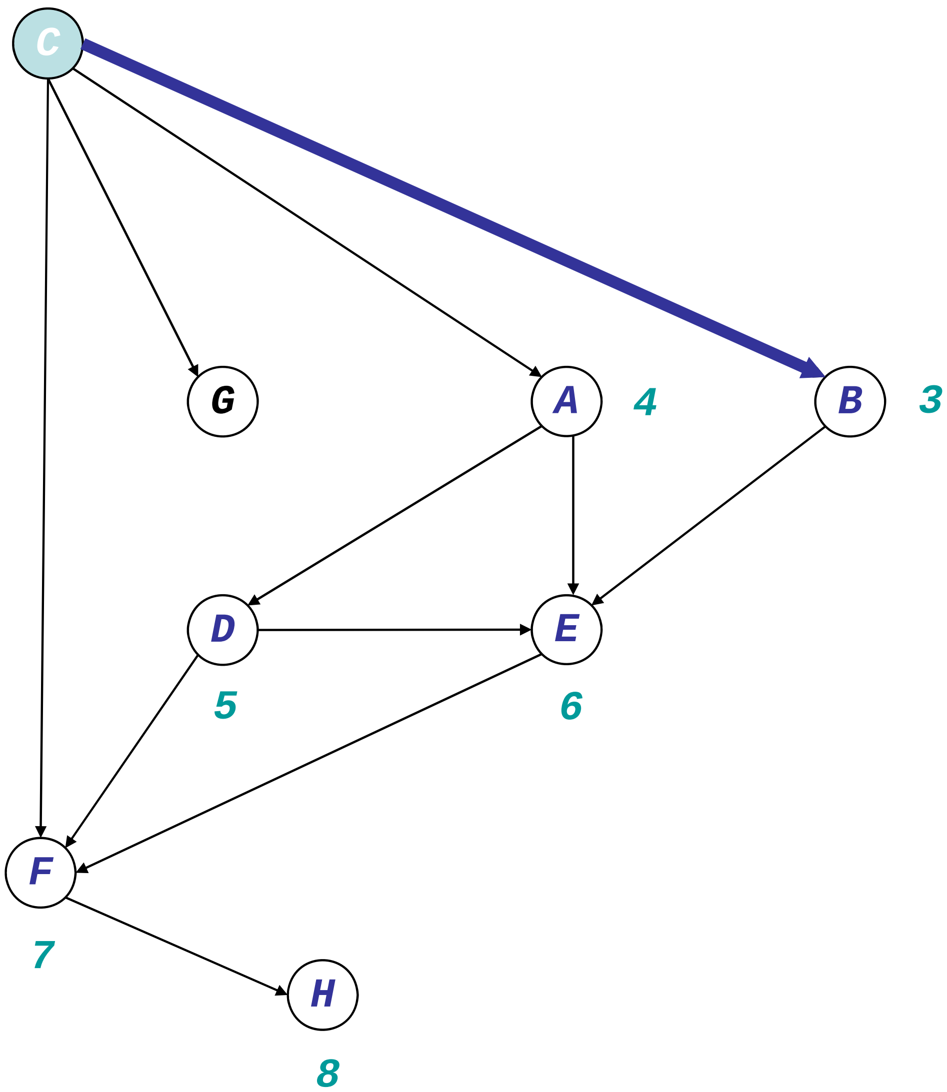


Topological Sort: DFS



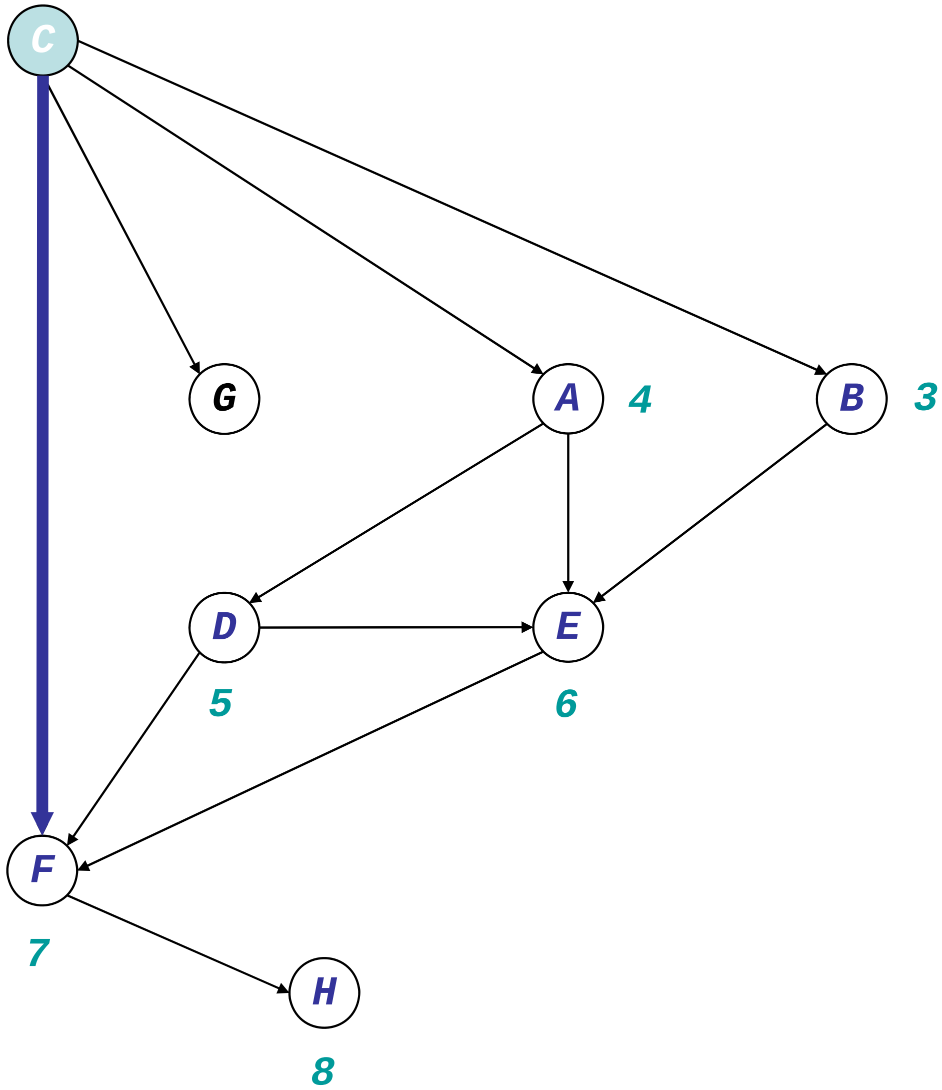
`dfs(C)`

Topological Sort: DFS



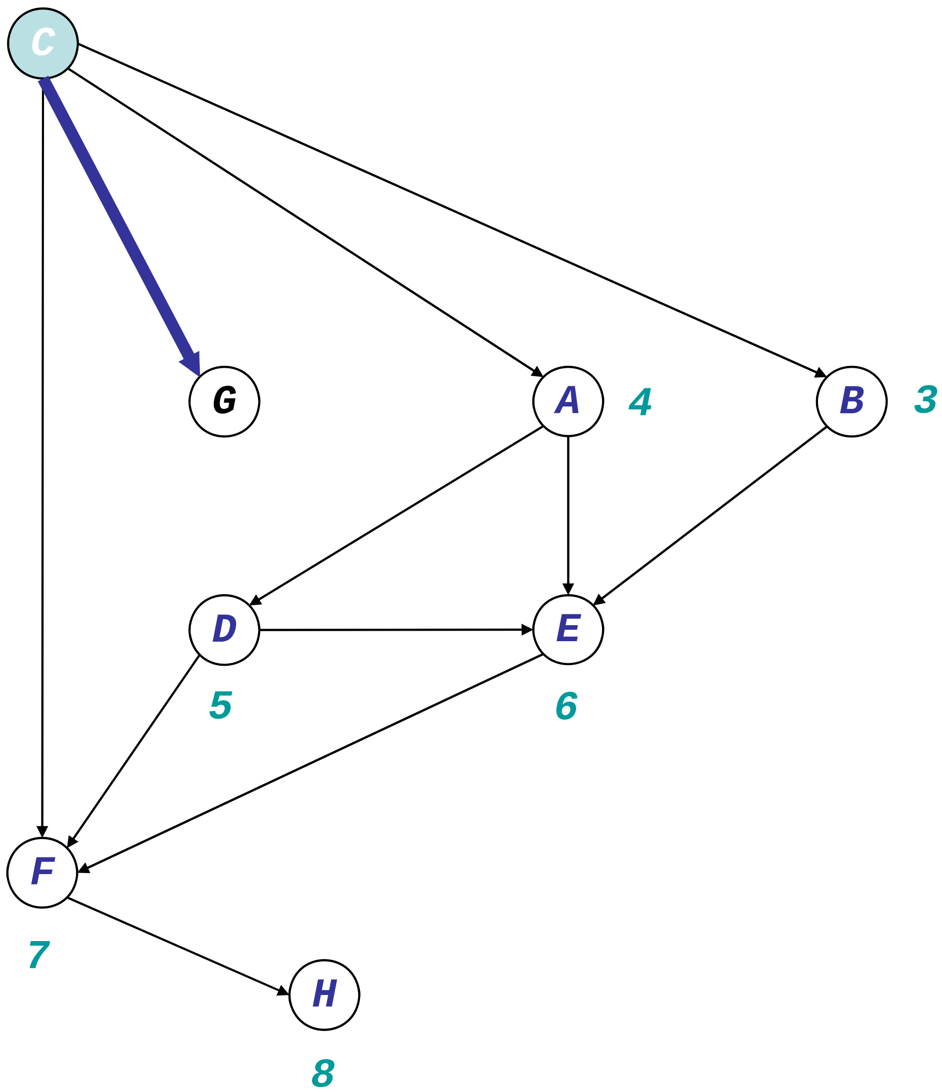
`dfs(C)`

Topological Sort: DFS



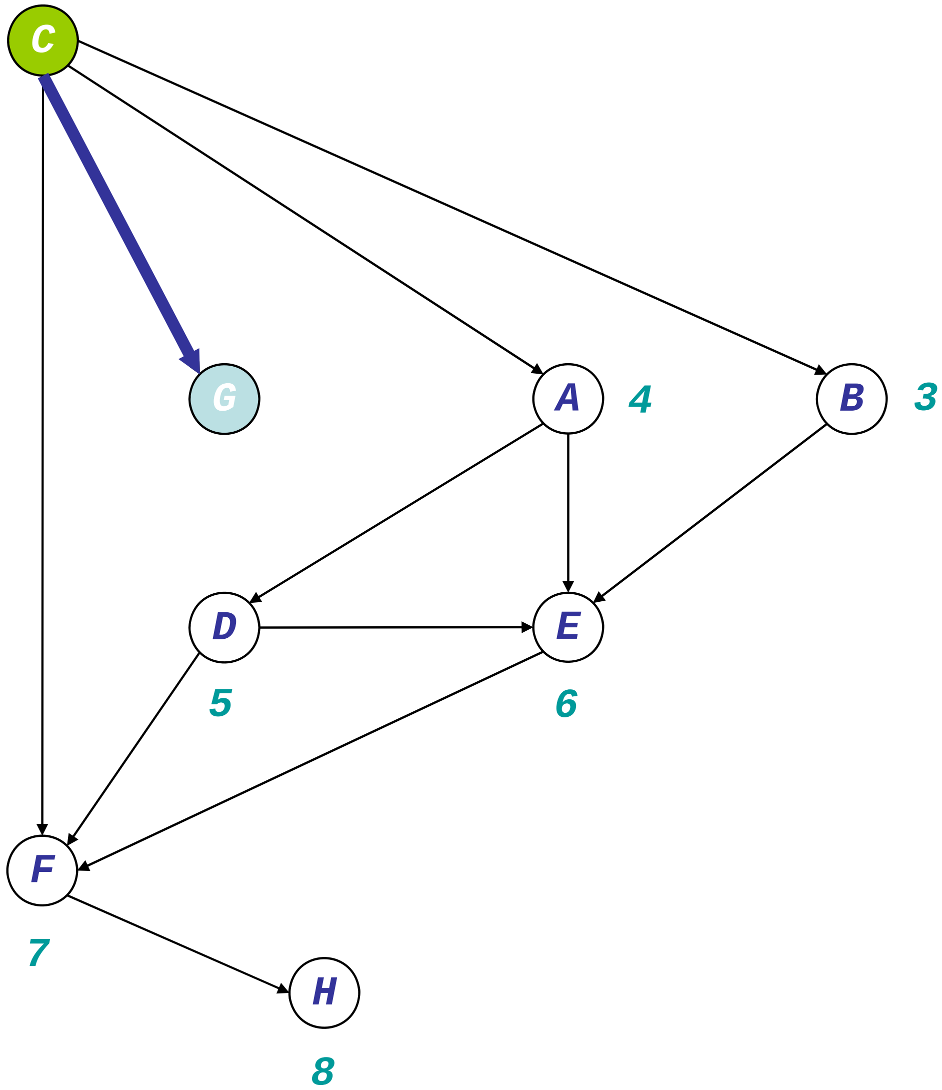
`dfs(C)`

Topological Sort: DFS



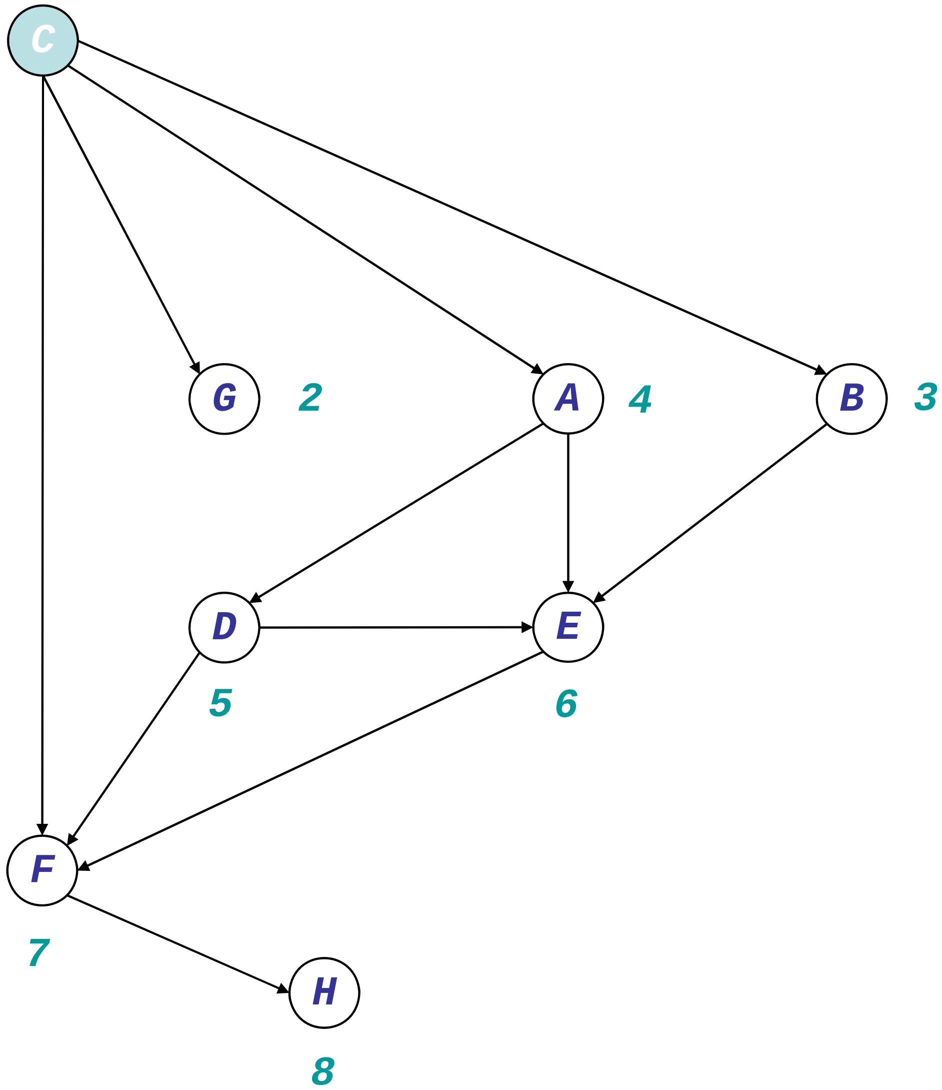
`dfs(C)`

Topological Sort: DFS




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

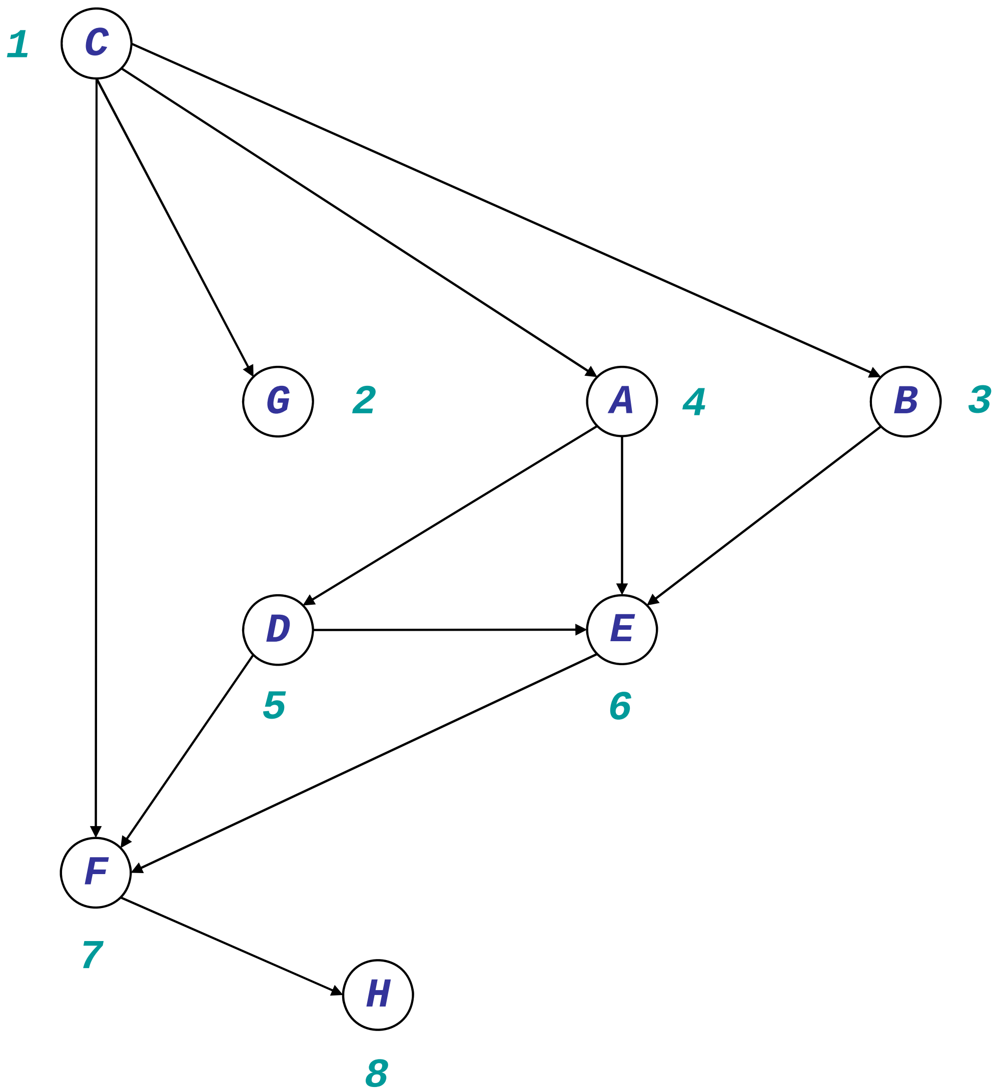
Topological Sort: DFS



`dfs(C)`

A/P Huang Guangbin

Topological Sort: DFS

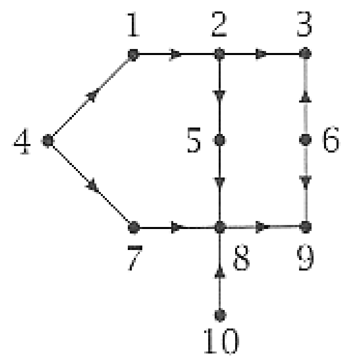


Topological order: C G B A D E F H

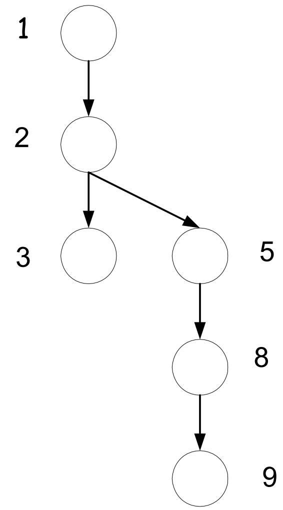
An Application of Topological Sorting

```
Topological_sort(adj)  {
    n = adj.last
    k = n // k is 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,i)
}
dfs_recurs(adj,v) {
    visit v
    visit[v] = true
    u = adj[v]
    while (u!= null) {
        if (!visit[u])
            dfs_recurs(adj, u)
        u = u.next
    }
    ts[k] = v
    k = k - 1
}
```

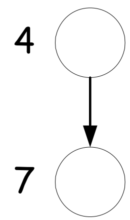
Input Graph



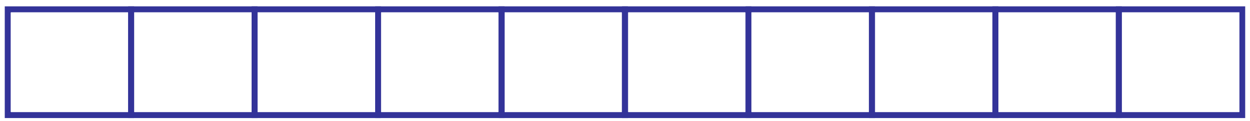
dfs tree



dfs tree



Topological sort array: ts

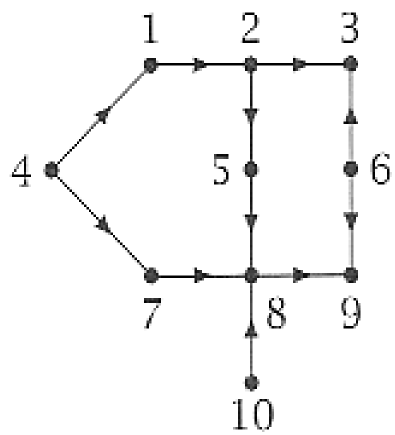


An Application of Topological Sorting

- **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*

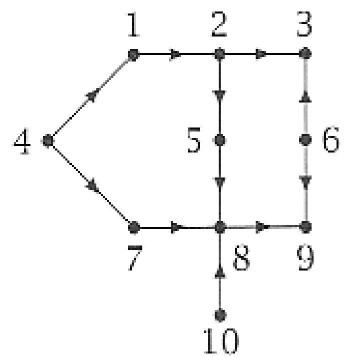
1 : MATH130
2: MATH 140
3: MATH 200
4: MATH 120
5: COMPSCI 150
6: PHYS 130
7: COMPSCI 100
8: COMPSCI 200
9: COMPSCI 240
10: ENG 110



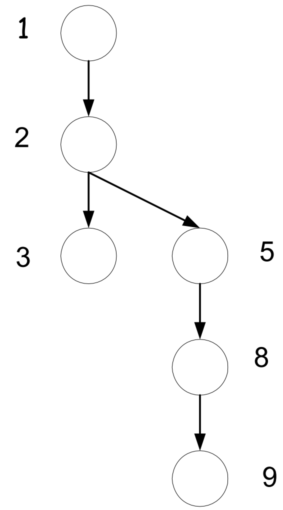
An Application of Topological Sorting

```
Topological_sort(adj)  {
    n = adj.last
    k = n  // k is 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,i)
}
dfs_recurs(adj,v) {
    visit v
    visit[v] = true
    u = adj[v]
    while (u!= null) {
        if (!visit[u])
            dfs_recurs(adj, u)
        u = u.next
    }
    ts[k] = v
    k = k - 1
}
```

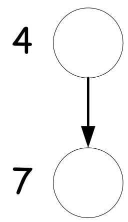
Input Graph



dfs tree



dfs tree



Topological sort array: ts

10	6	4	7	1	2	5	8	9	3

A/P Huang Guangbin

Searching Methods

- ☑ **Elementary searching methods**
  - ☞ **Sequential (Linear) search**
  - ☞ **Binary search**
- ☑ **Graph Search Algorithms**
  - ☞ **Breadth-First Search (BFS)**
  - ☞ **Depth-First Search (DFS)**
- ☐ **Text Searching Algorithms**

Algorithm Design Techniques

Algorithm Design Techniques

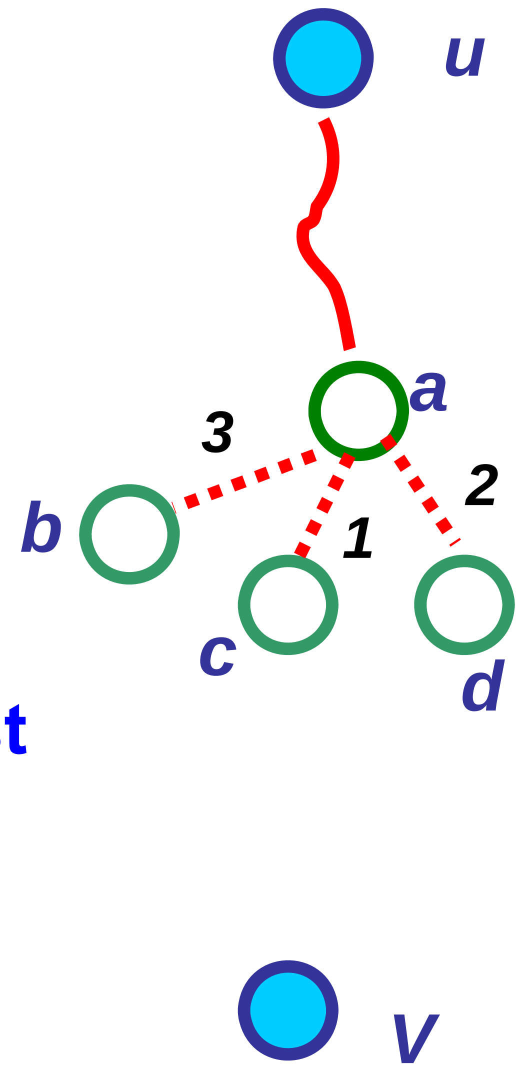
- ❑Fundamental algorithmic design techniques
  - ☞Fundamental => can be applied to a wide variety of algorithm design problems

    - ✓Recursive methods
    - ✓Divide and Conquer methods
    - ✓Greedy Algorithms

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)

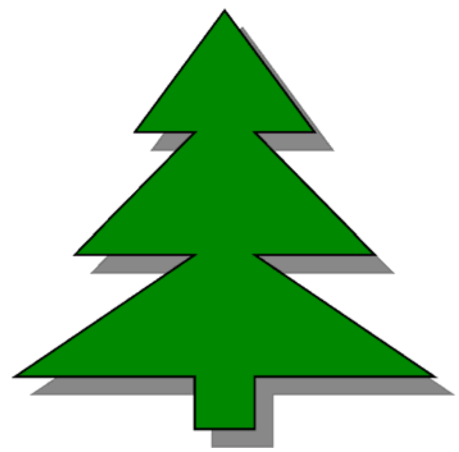


Applications of the Greedy Technique

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

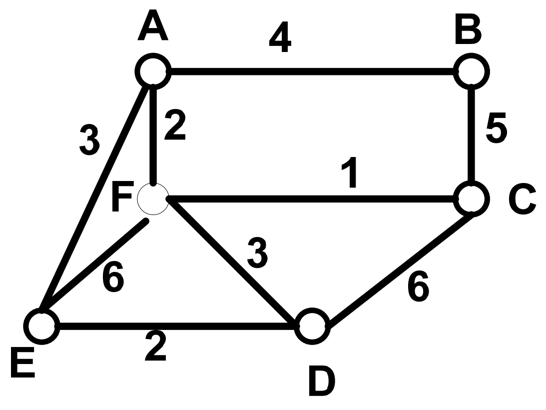
- **Shortest Path**
  - ☞ **Dijkstra's Algorithm**

Minimum Spanning Trees

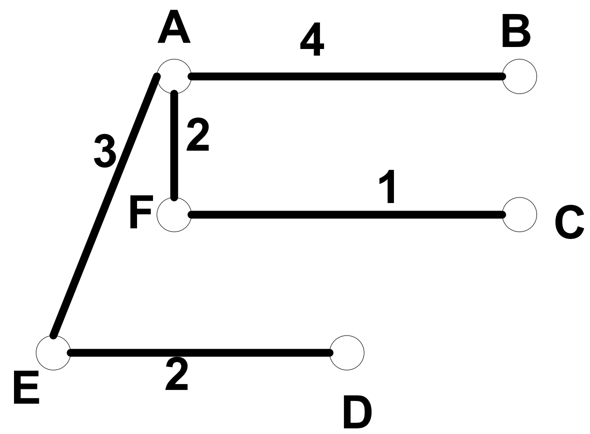


Minimum Spanning Trees

- 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*.

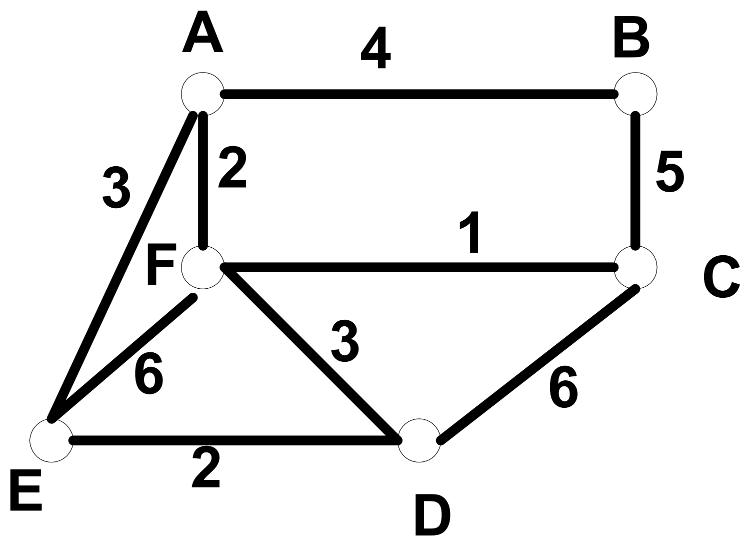


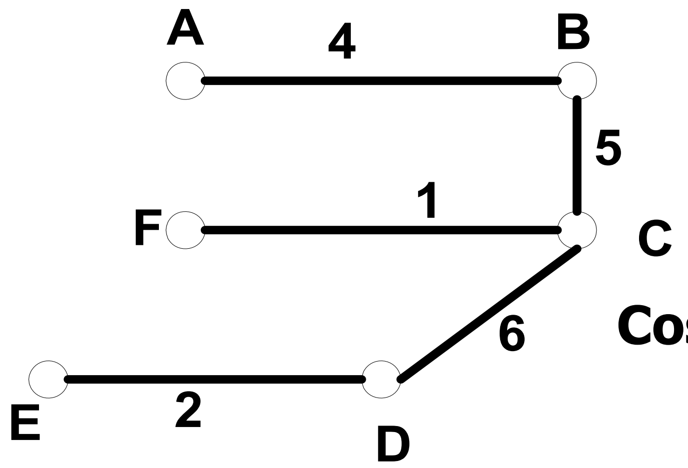
A weighted graph



A minimum spanning tree (weight = 12)

An Application of MST



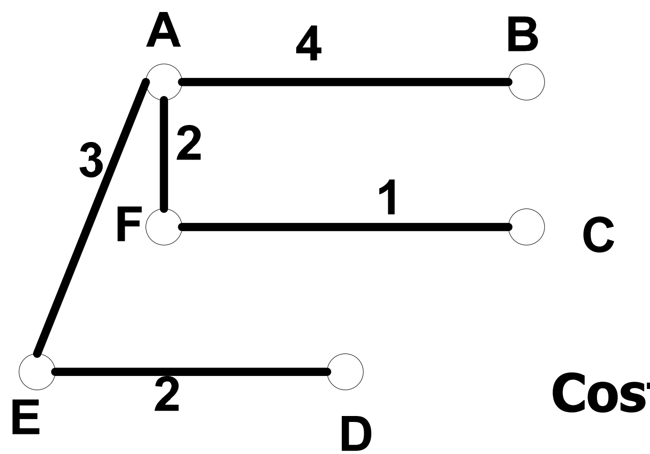


Cost = 18

Each node represents a city

Weight of each edge: cost of building a road connecting two cities ($b)

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



Cost = 12

Constructing MST

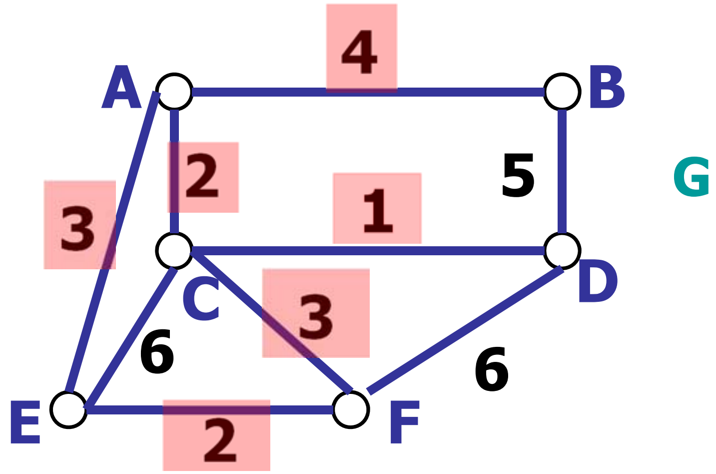
- A Minimum Spanning Tree can be constructed using one of the following algorithms
  - Kruskal's Algorithm
  - Prim's Algorithm

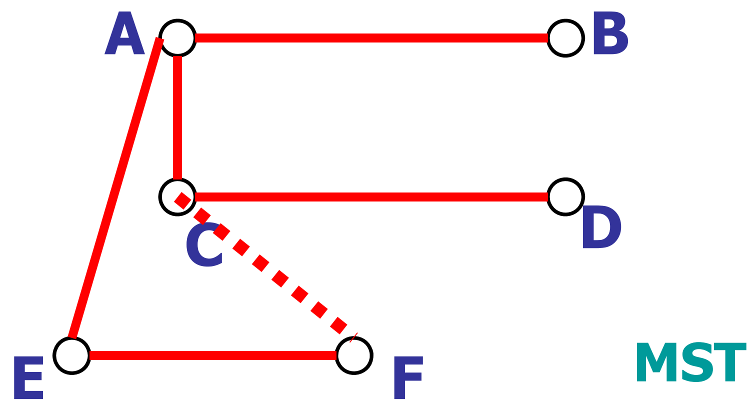
Kruskal's Algorithm

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



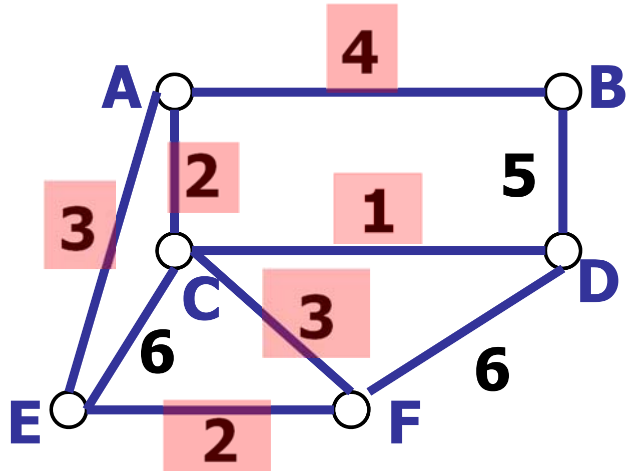


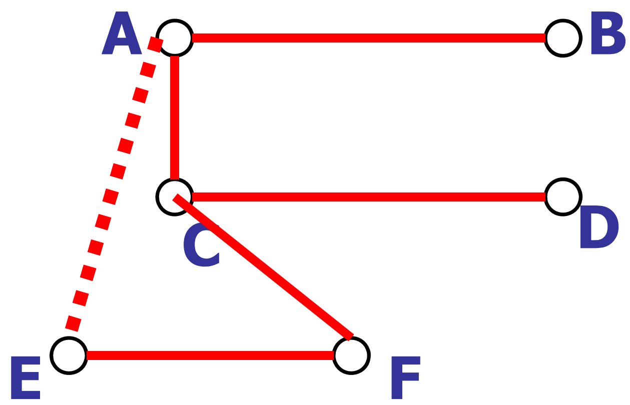
MST need not be a tree until the completion of Kruskal's Algorithm

Kruskal's Algorithm

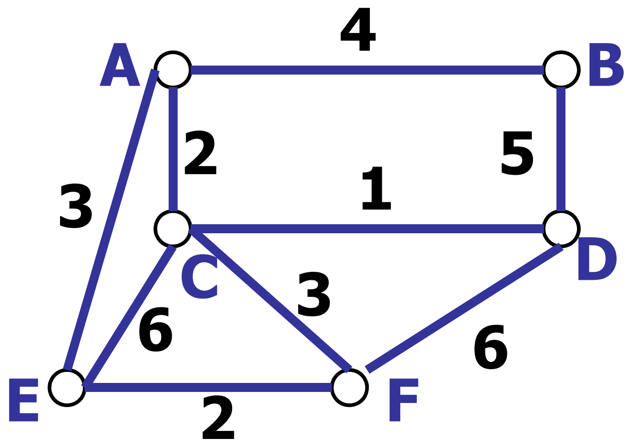
<u>Kruskal's Algorithm</u>

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

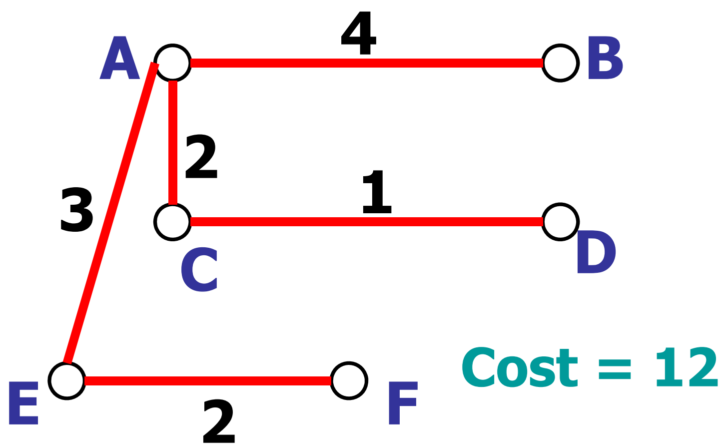


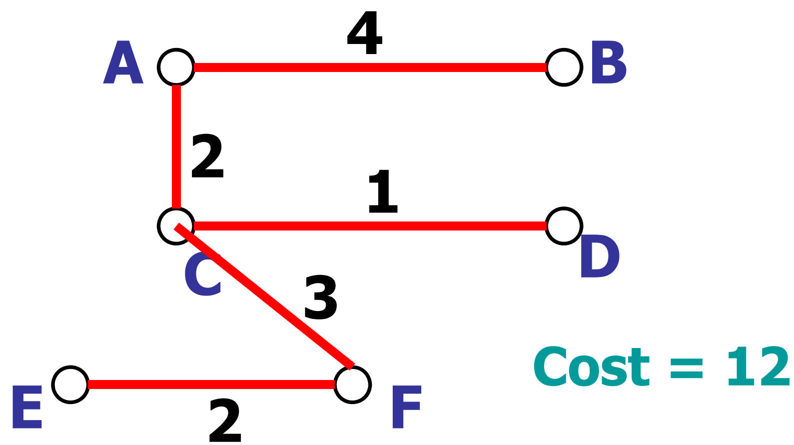


Kruskal's Algorithm



Minimum Spanning Trees are unique?

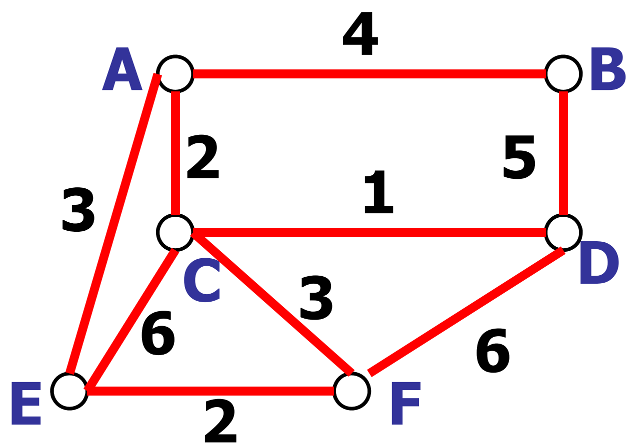




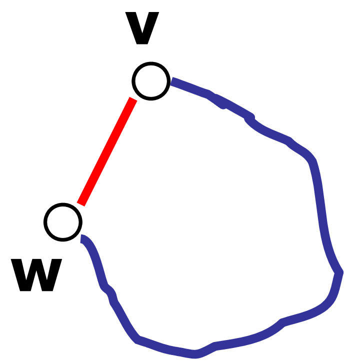
Implementation of Kruskal's Algorithm

- **Graph Representation**
  - (v1, v2, w): edge (v1 , v2) of weight w

- **Sort edges in non-decreasing order by weight**
  - (C,D,1) (A,C,2) (E,F,2) (A,E,3) (C,F,3) (A,B,4) (B,D,5) (C,E,6) (D,F,6)



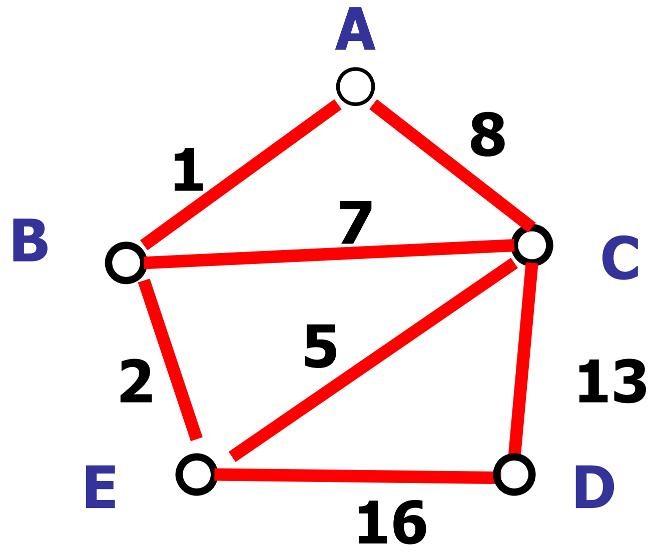
- **Edge selection**
  - Select edge with least weight for inclusion into the tree
  - Need to ensure selected edge will not create a cycle
    - ✓ Adding edge (v,w) will create a cycle when
      - ❖ there is a path between v and w formed by edges already selected
      - ❖ v and w belongs to the same connected component

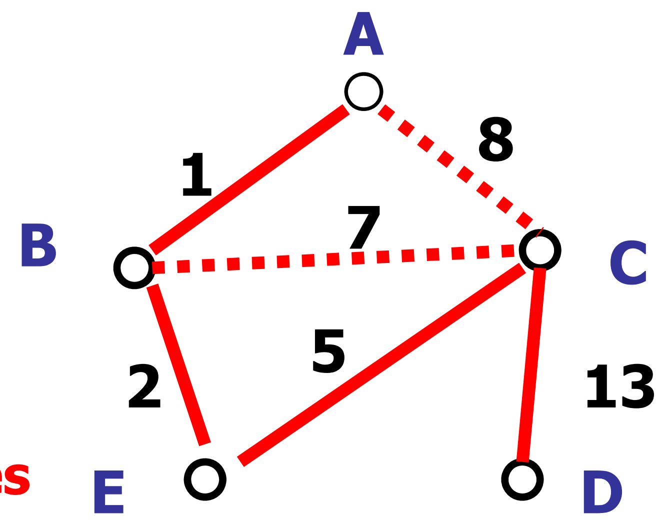


Kruskal's Algorithm

<u>Kruskal's Algorithm</u>

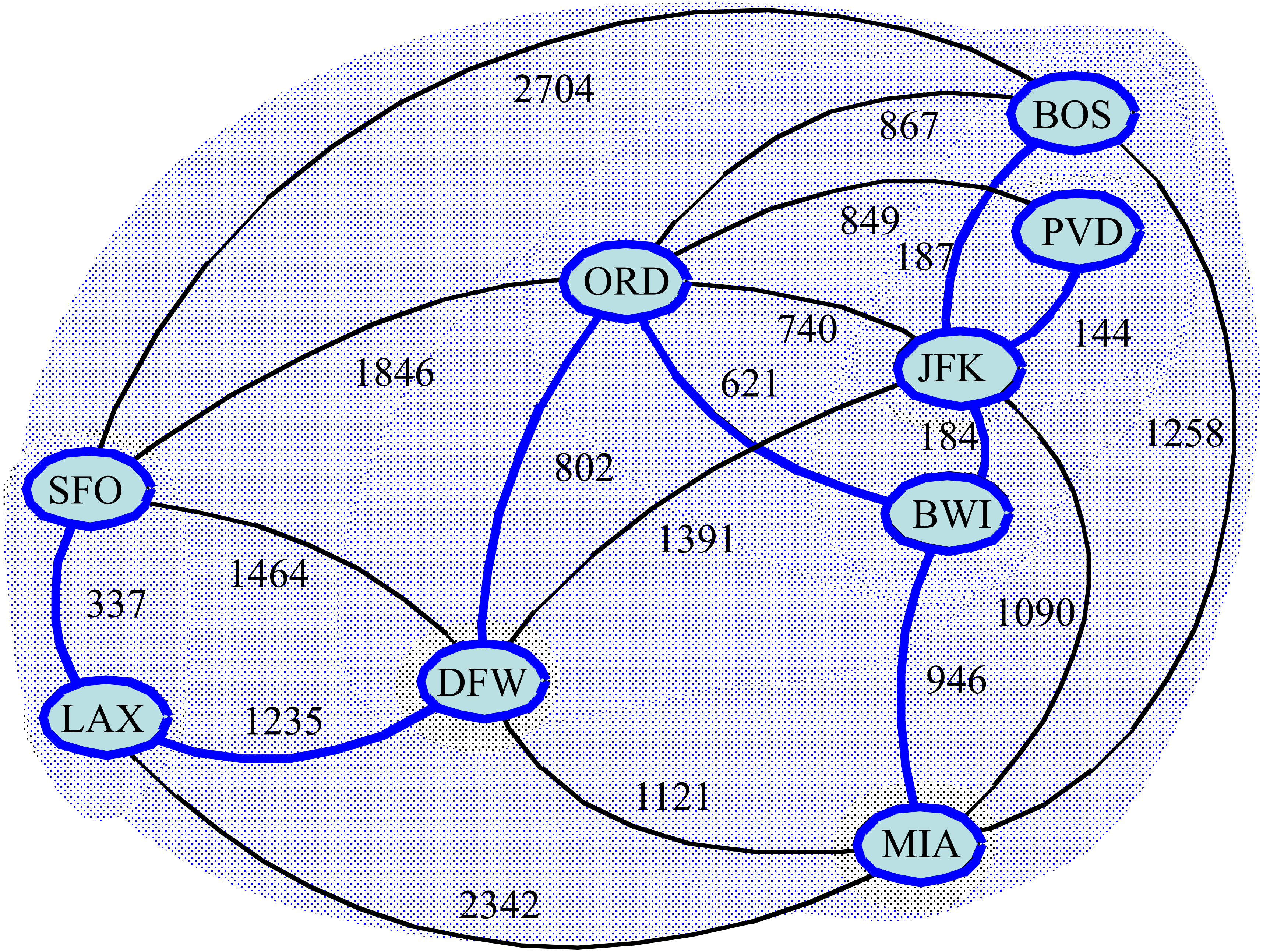
1) Add all vertices of G in the MST
2) Add an edge of minimum weight of G to the MST
3) If the number of edges of MST is less than n-1, repeatedly add an edge of next minimum weight of G that does not make a cycle to the MST





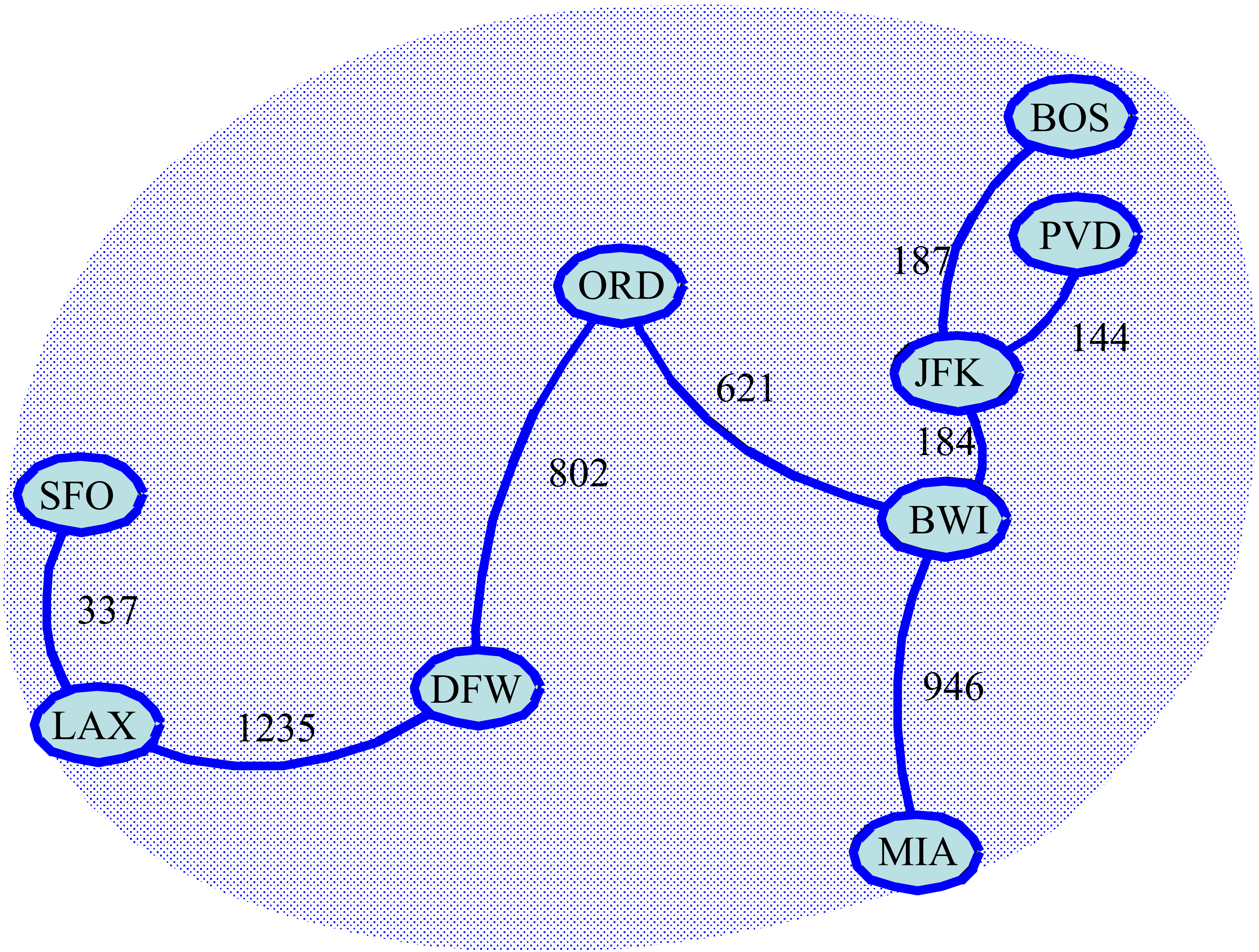
Are Minimum Spanning Trees of this graph unique?

Example



A/P Huang Guangbin

Example



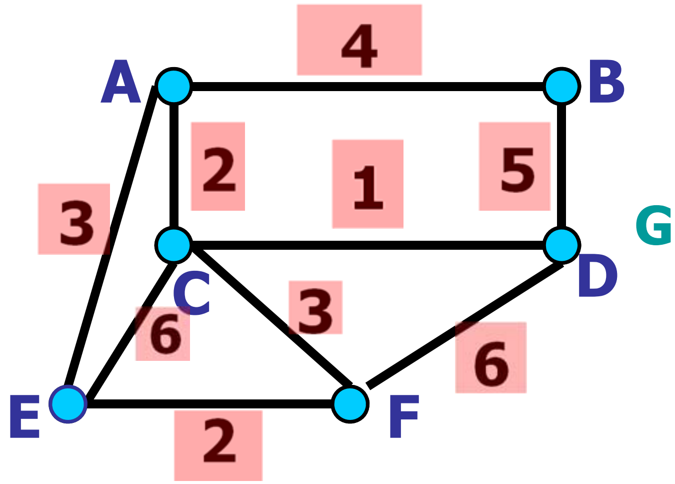
BOS
PVD
187
ORD
144
JFK
621
184
802
SFO
BWI
337
946
DFW
LAX
1235
MIA

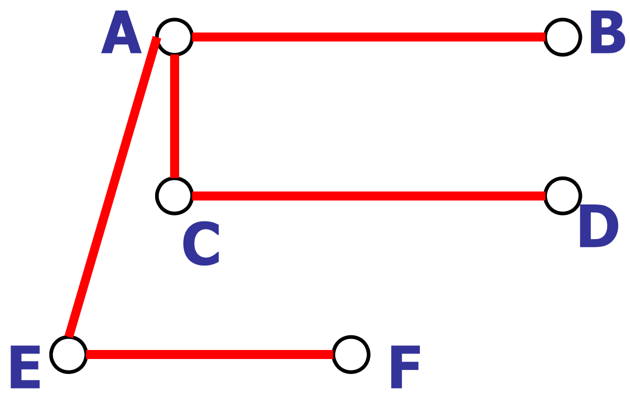
Prim’s Algorithm

Prim’s Algorithm

Prim’s Algorithm

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

Prim's Algorithm: Pseudo Code

$V_T \leftarrow \{v_0\}$ // V_T is the set of vertices in MST

$E_T \leftarrow \phi$ // E_T is the set of edges in MST

for i = 1 **to** n-1 **do** {

find minimum-weight e=(u,v) where $u \in V_T$ and $v \in V - V_T$
// V is the set of vertices in the original graph G

$V_T \leftarrow V_T \cup \{v\}$

$E_T \leftarrow E_T \cup \{e\}$

}

Time Complexity Prim's Algorithm

$V_T \leftarrow \{v_0\}$ **// V_T is the set of vertices in MST**

$E_T \leftarrow \phi$ **// E_T is the set of edges in MST**

for i = 1 to n-1 do { **O(n)**

find minimum-weight e=(u,v) where u $\in V_T$ and v $\in V - V_T$
// V is the set of vertices in the original graph G **O(m)**

$V_T \leftarrow V_T \cup \{v\}$

$E_T \leftarrow E_T \cup \{e\}$

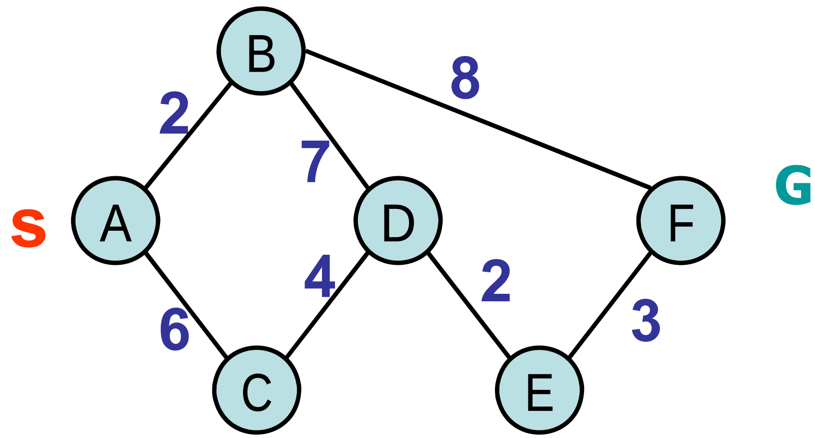
}

Overall complexity = O(nm)

O(n)* O(m) <= k_1n *k_2m <= k_1k_2 * nm

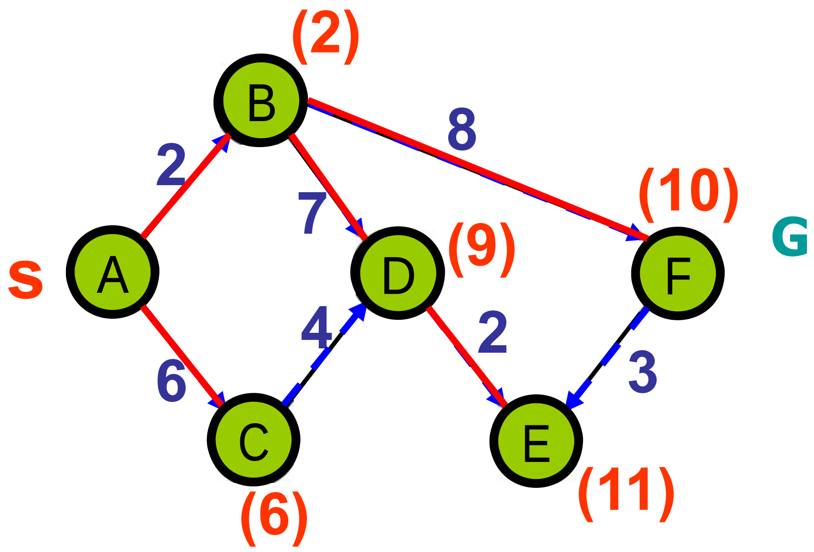
Shortest Paths

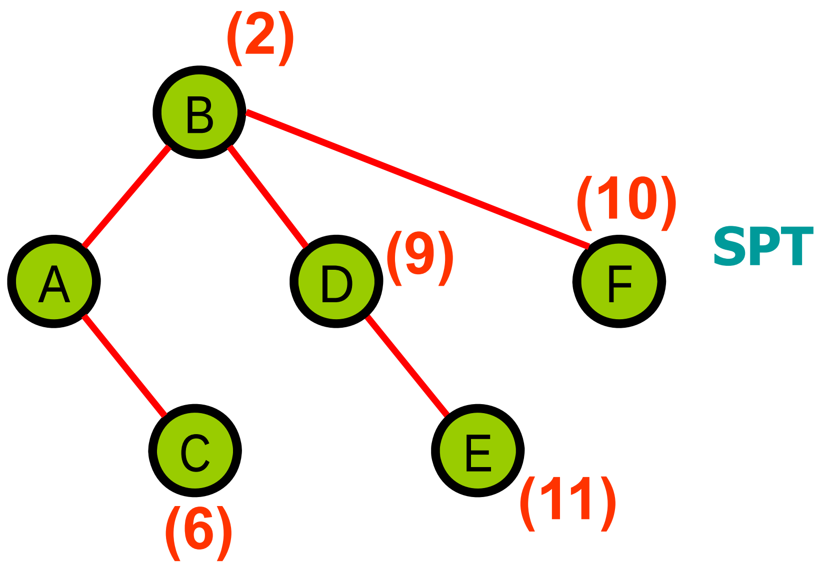
Shortest Path Problem



- **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



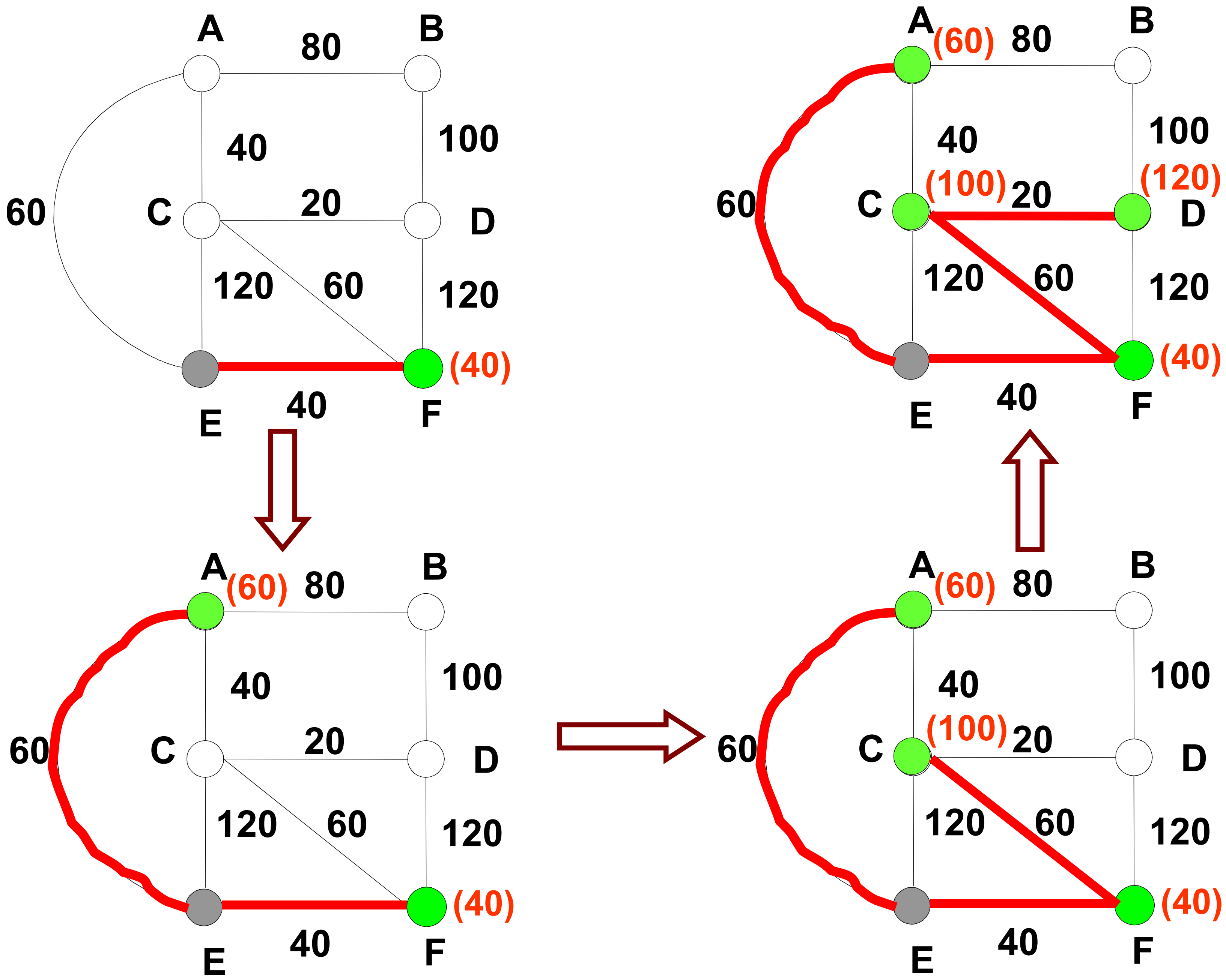


Dijkstra's Algorithm (s)

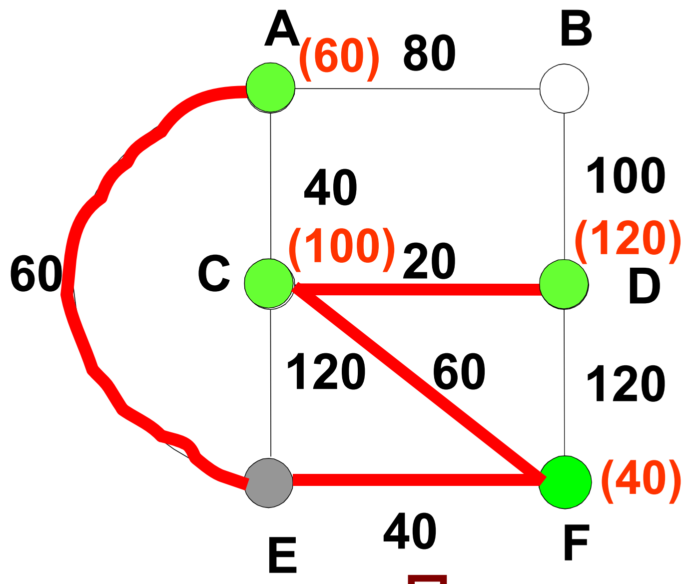
1) **Add a minimum edge starting from s to an empty SPT**
2) **If the number of the edges of SPT is less than n-1, keep growing tree SPT by repeatedly adding edges which can extend the paths from s in SPT as short as possible.**

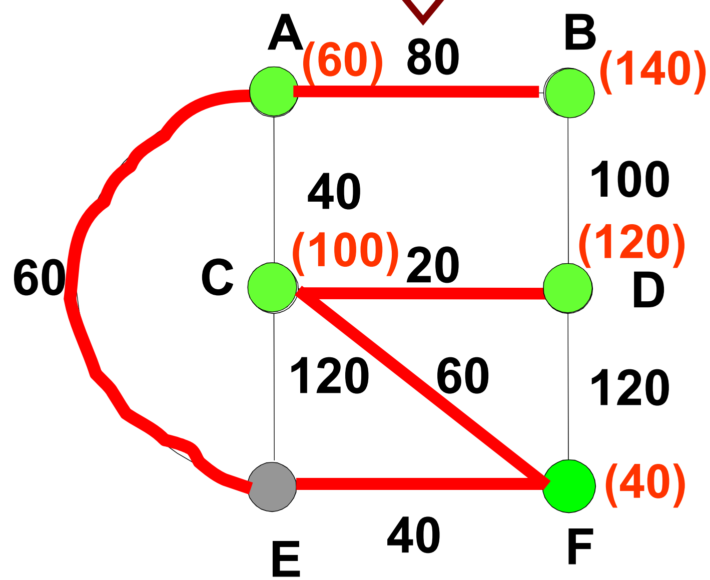
SPT always remains as a tree when Dijkstra's Algorithm runs

Example


A
80
B
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
60
C
(100)
20
(120)
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

```
VT  = {1};  k = 0;  s = 1;
for j = 1 to n                                   O(n)
     dist[j] = w(1,j)
While k < n  do{                                 O(n)
     min = ∞
     for each j ∈ V - VT  do{                    O(n)
          if dist[j]  < min then{
               min = dist[j]
               new = j
          }
     }
     VT = VT ∪ {new}
     k = k + 1
     for j ∈ V - VT  do{                         O(n)
          if dist[new] + w(new, j)  <  dist[j]{
               dist[j]  = dist[new] + w(new, j)
          }
     }
}
```

Overall complexity = $O(n^2)$

Inductive Proof of Dijkstra's Algorithm

Conditions: all the weights are non-negative

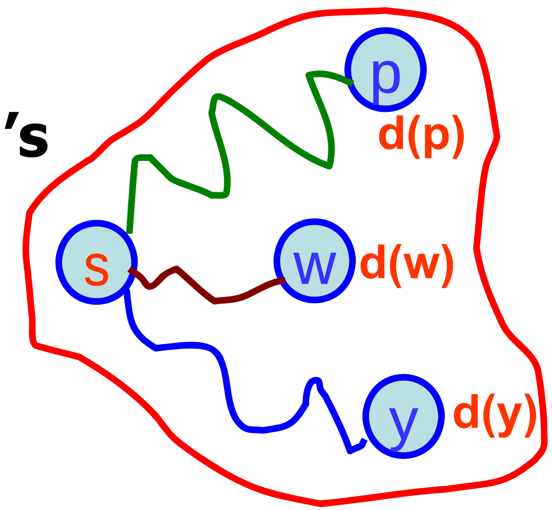
Basis Step:

If n=1, the tree contains zero edge

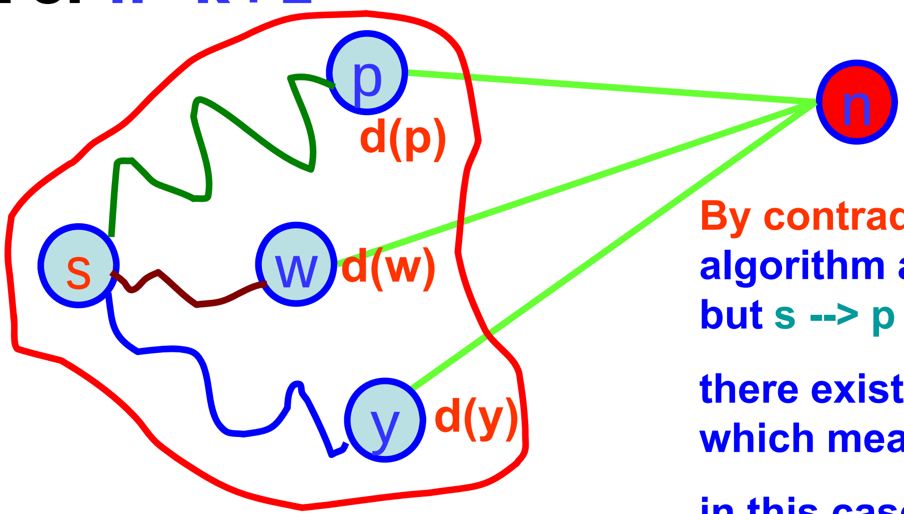
If n=2, the tree contains one edge

Inductive Step:

Assume graph G has n<=k vertices, Dijkstra's algorithm obtains the right shortest paths



For n=k+1



By contradiction method: suppose according to Dijkstra's algorithm a newly added vertex n by the path s --> p --> n, but s --> p --> n is not the shortest path from s --> n

there exists a shortest path from s --> n, say, by vertex y, which means d(y)+L(y,n)<d(p)+L(p,n),

in this case, by Dijkstra's algorithm edge e=(y,n) should be added instead of e=(p,n), which makes the contradiction

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**
    - **Use Dijkstra's Algo**
      - **Complexity $O(n^2)$**

A/P Huang Guangbin

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**

Examination

- **Duration**
  - 2.5 hours
- **Format**
  - Answer 4 questions (no choice)
- **Coverage**
  - LCP (1.75 Questions each), HGB or TWP (2.25 Questions each)
  - Evenly distributed
- **Expectations**
  - take tutorial questions, CA & past years' exam papers as a guide