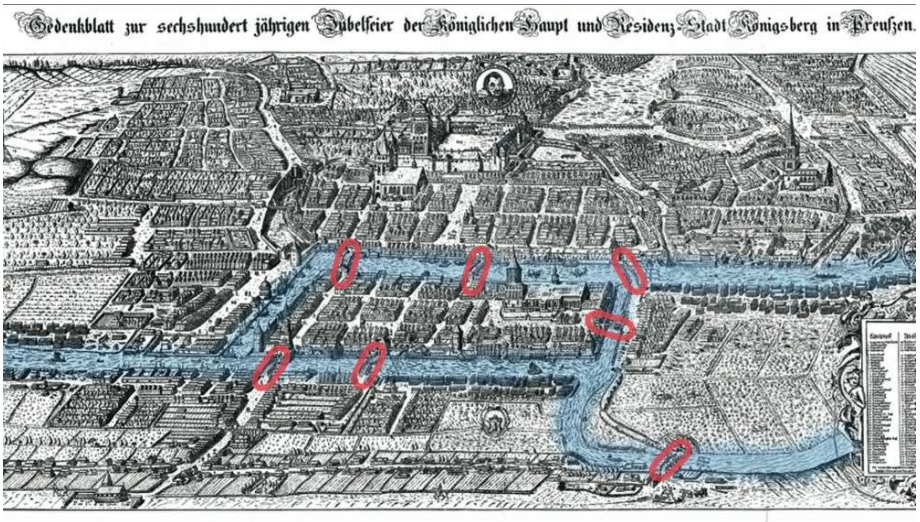
CSIEB0100 Data Structures

Lecture 07 Graphs

Shiow-yang Wu 吳秀陽
Department of Computer Science and Information Engineering
National Dong Hwa University

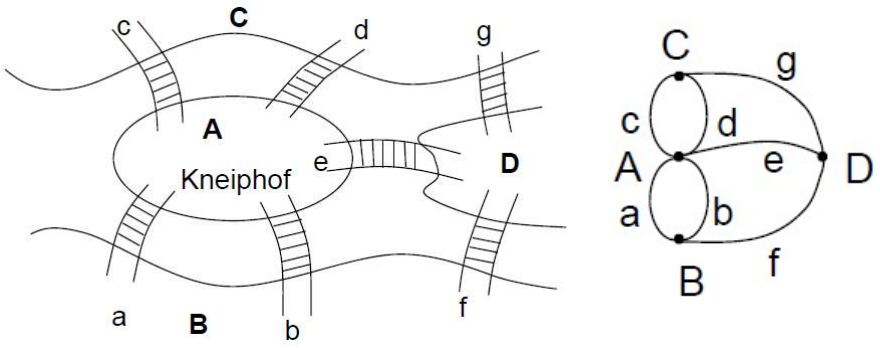
Lecture material is partly home-grown, partly taken from slides came with the textbook originally prepared by Professor Jiun-Long Huang of NCTU.

The City of Konigsberg



Konigsberg Bridge Problem

- Can the seven bridges of the city of Königsberg be traversed in a single trip without doubling back (the trip ends in the same place it began).



Euler's Graph

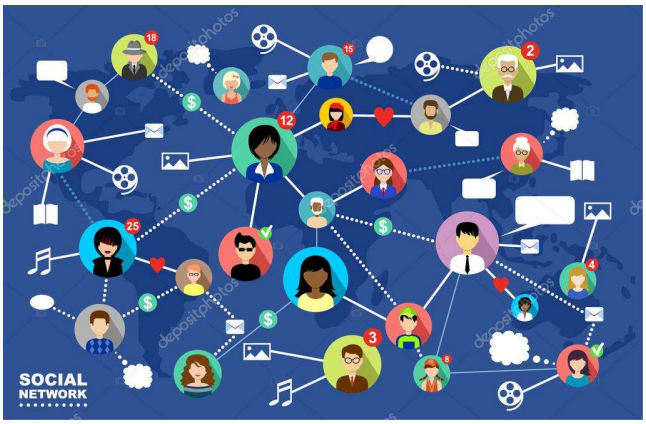
- The resolution of the problem by Leonhard Euler in 1736 laid the foundations of graph theory and prefigured the idea of topology.
- Degree of a vertex: no. edges incident to it.
- Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even. This is called an Eulerian walk(Eulerian cycle).
- No Eulerian walk of the Konigsberg bridge problem since all four vertices are of odd edges.

The Rise of Social Media

- Social media is one of the greatest invention of 21st century. (Facebook, YouTube, WhatsApp, Instagram, WeChat, TikTok, Twitter, …)
- More than 4.7 billion people (~60% of the world's population) use social media. (2023)
- Many services on social media are based on analyzing and exploring the social relationships among users.
- Social relationships are best represented by social graph. (next slide)

Social Graph

- A social graph is a diagram that represents the social relationships between entities.



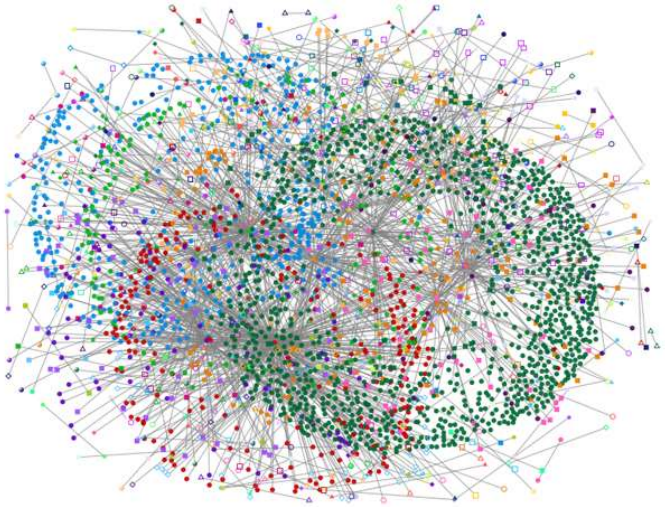
(https://cn.depositphotos.com/vector-images/social-networks.html)

Application of Graphs

- Analysis of electrical circuits
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Statistical mechanics
- Genetics
- Cybernetics
- Linguistics
- Social Sciences

Big Graph Analytics

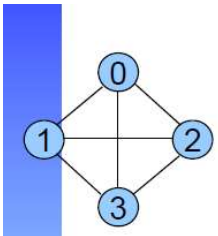
- Real-world graphs can be HUGE !!
- Big graph analytics is a hot area of research.
- Even used in fighting terrorism!



Definition of a Graph

- A graph, G=(V, E), consists of two sets, V and E.
  - V is a finite, nonempty set of vertices.
  - E is set of pairs of vertices called edges.
- The vertices of a graph G can be represented as V(G).
- Likewise, the edges of a graph, G, can be represented as E(G).
- Graphs can be either undirected graphs or directed graphs.
- For a undirected graph, a pair of vertices *(u, v)* or *(v, u)* represent the same edge.
- For a directed graph, a directed pair <*u, v*> has *u* as the tail and the *v* as the head. Therefore, <*u, v*> and <*v, u*> represent different edges. $u \longrightarrow v$

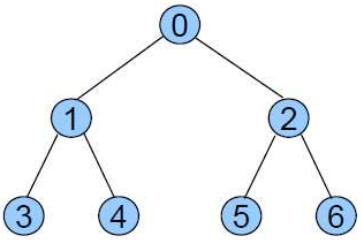
Three Sample Graphs



$V(G_1) = \{0, 1, 2, 3\}$

$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$

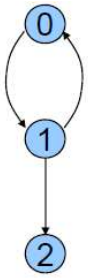
(a) G_1



$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$

$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$

(b) G_2



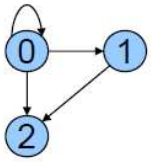
$V(G_3) = \{0, 1, 2\}$

$E(G_3) = \{<0, 1>, <1, 0>, <1, 2>\}$

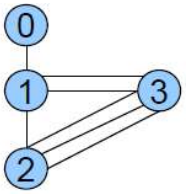
(c) G_3

Graph Restrictions

- A graph may not have an edge from a vertex back to itself
  - *(v, v)* or <*v, v*> are called self edge or self loop.
- A graph may not have multiple occurrences of the same edge
  - Without this restriction, it is called a multigraph.



(a) Graph with a self edge



(b) Multigraph

Terminologies of Graph

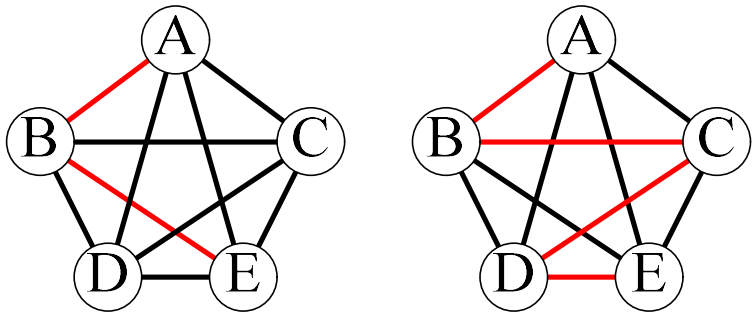
- Graph: G=(V, E)
- V: a set of vertices (vertex set)
- E: a set of edges (edge set)
- Edge (arc): A pair *(v,w)*, where $v, w \in V$
- Directed graph (Digraph): A graph with ordered pairs (directed edge)
- Adjacent: w is adjacent to *v* if $(v,w) \in E$
- Undirected graph: If $(v,w) \in E$, $(v,w)=(w,v)$

Terminologies of Graph

- Path: a sequence of vertices w_1, w_2, w_3,…, w_N where $(w_i, w_{i+1}) \in E$, $\forall\, 1 \le i \le N$.
- Length of a path: number of edges on the path.
- Simple path: a path where all vertices are distinct except the first and last.
- Cycle in a directed graph: a path such that $w_1 = w_N$
- Acyclic graph (DAG): a directed graph with no cycle.

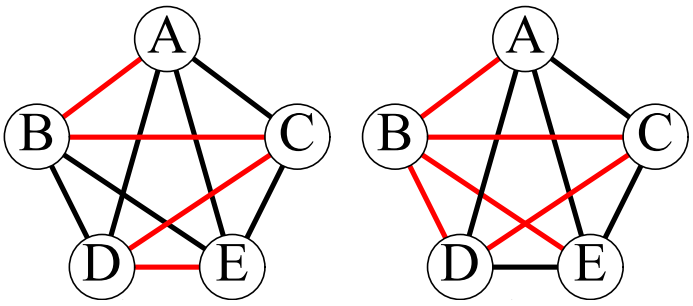
Paths (Examples)

- In the following graph, both {(A,B), (B, E)} and {(A,B), (B, C), (C, D), (D, E)} are paths.



Simple Paths

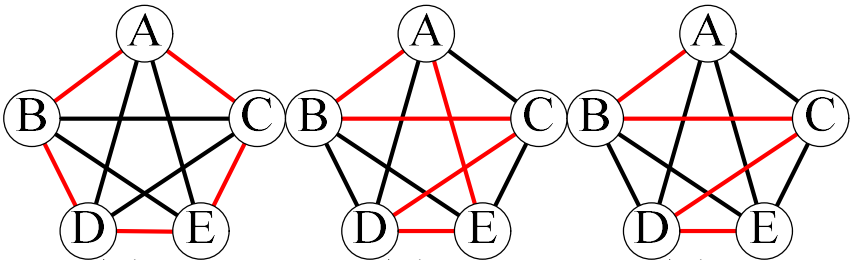
- In the following graph, the path on the left is a simple path while the path on the right is not a simple path.



Simple path　　Not a simple path

Cycles

- Here are some examples of cycles and non-cycles.



Simple path
Cycle

Simple path
Cycle

Simple path
Not a cycle

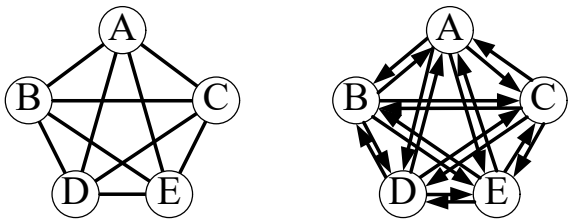
Terminologies of Graph

- Connected (for an undirected graph): if there is a **path** from every vertex to every vertex.
- Strongly connected (for a directed graph): if there is a **path** from every vertex to every vertex.
- Complete graph: a graph in which there is an **edge** between every pair of vertices. (next slide)
- The density of a graph is the ratio between the number of edges and the maximum possible edges. i.e.

  (density) = (# edges) / (# possible edges)

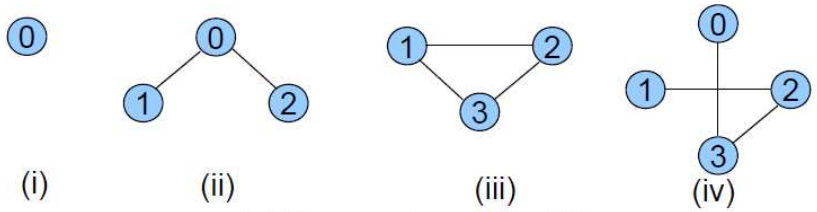
Complete Graphs

- The number of distinct unordered pairs *(u, v)* with *u≠v* in a graph with n vertices is *n(n-1)/2.*
- A complete undirected graph is an undirected graph with exactly *n(n-1)/2* edges.
- A complete directed graph is a directed graph with exactly *n(n-1) edges.*

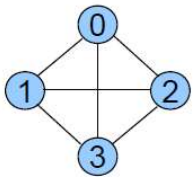


Subgraphs

- A subgraph of G is a graph G' such that V(G')⊆V(G) and E(G')⊆E(G).

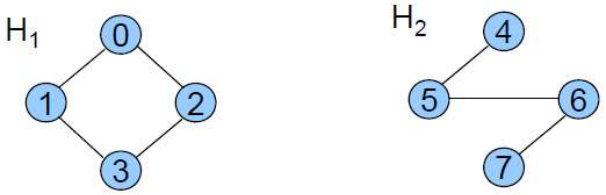


(a) Some subgraphs of G_1



Connected Components

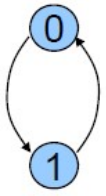
- A connected component, H, of an undirected graph is a maximal connected subgraph.
  - By maximal, we mean that G contains no other subgraph that is both connected and properly contains H.
  - G_4 below is a graph with two connected components.



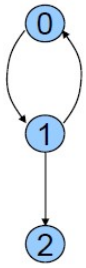
G_4

Strongly Connected Components

- A directed graph G is said to be strongly connected iff for each pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u.
- A strongly connected component is a maximal subgraph that is strongly connected.



Strongly Connected Components of G3



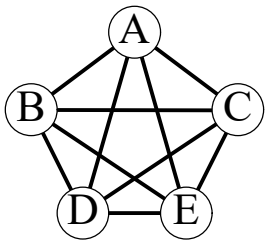
G3

Degree of a Vertex

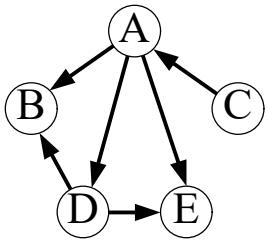
- The degree of a vertex is the number of edges incident to that vertex.
- If G is a directed graph, then we define
  - In-degree of a vertex: is the number of edges for which vertex is the head.
  - Out-degree of a vertex: is the number of edges for which the vertex is the tail.
- For a graph G with n vertices and e edges, if d_i is the degree of a vertex i in G, then the number of edges of G is $e = (\sum_{i=0}^{n-1} d_i) / 2$

Degree of a Vertex

- The degree of node A is 4.



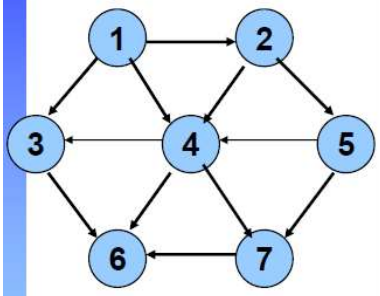
- The in-degree of node A is 1 while the out-degree is 3.



Abstract Data Type of Graphs

```
class Graph
{
/* objects: A nonempty set of vertices and a set of
   undirected edges where each edge is a pair of vertices */
public:
  Graph(); // Create an empty graph
  void InsertVertex(Vertex v);
  void InsertEdge(Vertex u, Vertex v);
  void DeleteVertex(Vertex v);
  void DeleteEdge(Vertex u, Vertex v);
  Boolean IsEmpty(); //if graph has no vertices return TRUE
  List<List> Adjacent(Vertex v);
  // return a list of all vertices that are adjacent to v
};
```

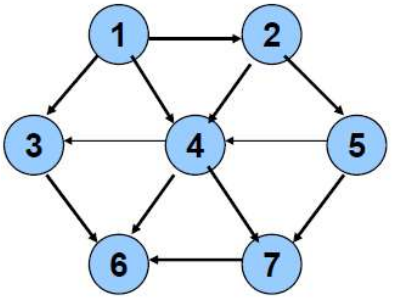
Adjacency Matrix Representation

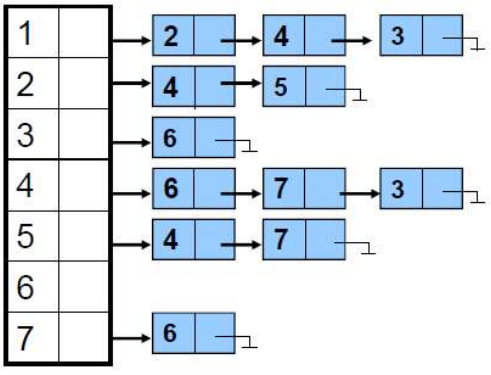


	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

- Space: $\Theta(|V|^2)$, good for dense, not for sparse
- Undirected graph: symmetric matrix (why?)

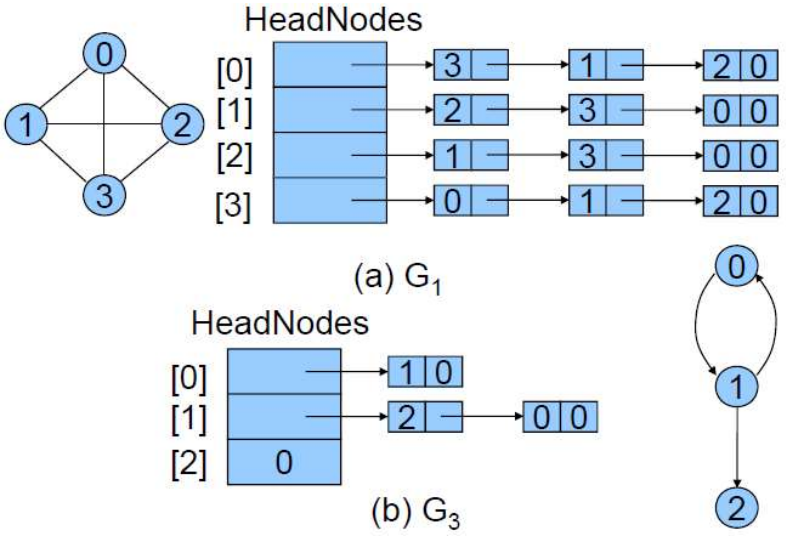
Adjacency List Representation



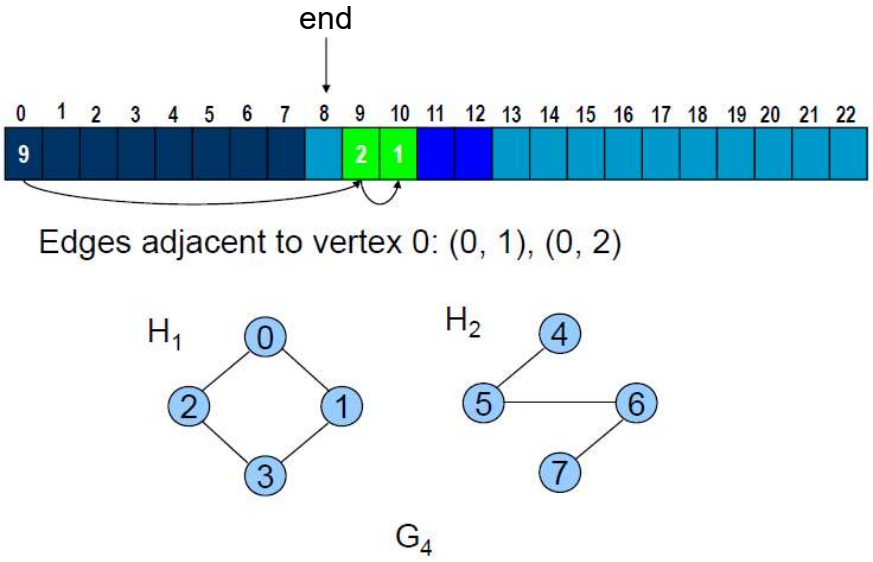


- Space: $O(|V|+|E|)$, good for sparse graphs

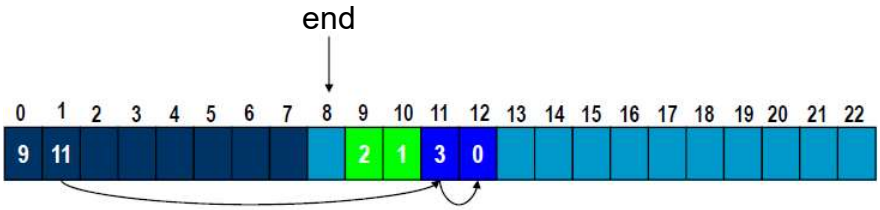
Adjacency Lists (Examples)



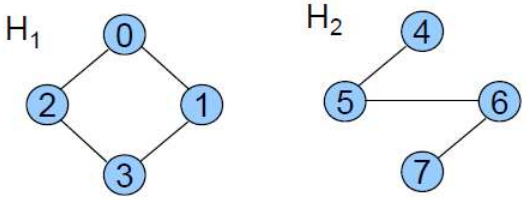
Sequential Representation of Graph G_4



Sequential Representation of Graph G_4

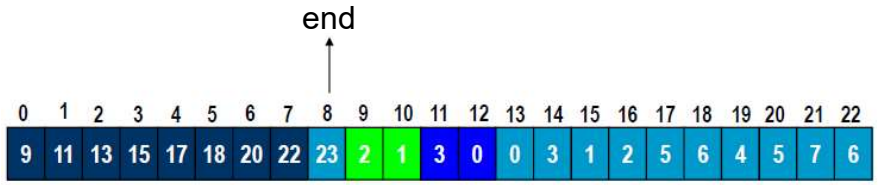


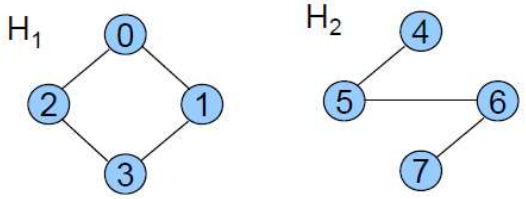
Edges adjacent to vertex 1: (1, 0), (1, 3)



G_4

Sequential Representation of Graph G_4





G_4

Inverse Adjacency Lists for G_3

- Adjacency list
  - Out-degree
- Inverse adjacency list
  - In-degree

Adjacency Multilists

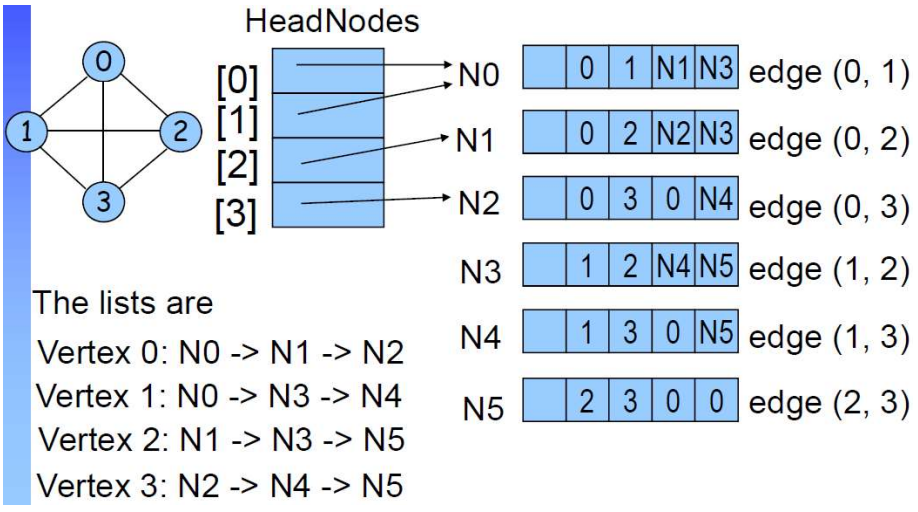
- In the adjacency-list representation of an undirected graph, each edge *(u, v)* is represented by two entries.
- Multilists: To be able to determine the second entry for a particular edge and mark that edge as having been examined, we use a structure called multilists (where nodes may be shared among several lists).
  - Each edge is represented by one node.
  - Each node will be in two lists.

Adjacency Multilists

- Each edge is represented by a node with the following structure

Marked Bit	V1	V2	Link1	Link2
One bit mark to represent whether the edge has been examined or not.	Starting vertex	Ending vertex	Next edge node of the starting vertex or NIL if none.	Next edge node of the ending vertex or NIL if none.

Adjacency Multilists for G_1



The lists are

Vertex 0: N0 -> N1 -> N2
Vertex 1: N0 -> N3 -> N4
Vertex 2: N1 -> N3 -> N5
Vertex 3: N2 -> N4 -> N5

Weighted Edges

- Very often the edges of a graph have weights associated with them.
  - Distance from one vertex to another
  - Cost of going from one vertex to an adjacent vertex
- To represent weight, we need additional field, `weight`, in each entry.
- A graph with weighted edges is called a *network*.

Graph Operations

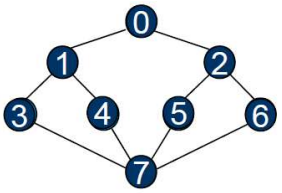
- A general operation on a graph G is to visit all vertices in G that are reachable from a vertex v.
  - Depth-first search
  - Breadth-first search
- Both search methods work on directed and undirected graphs.

Depth-First Search

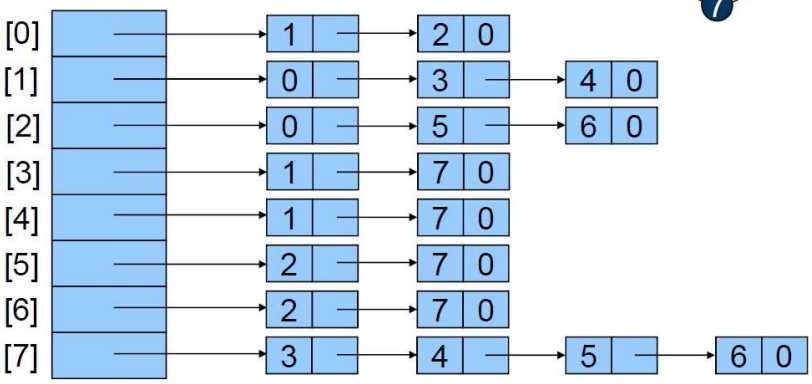
- Depth First Search(DFS): generalization of preorder traversal
- Starting from vertex *v*, process *v* & then recursively traverse all vertices adjacent to *v*.
  - Using stack
- To avoid cycles, mark visited vertices

DFS on Graph G(Adjacency Lists)

DFS from 0:
0, 1, 3, 7, 4, 5, 2, 6



HeadNodes



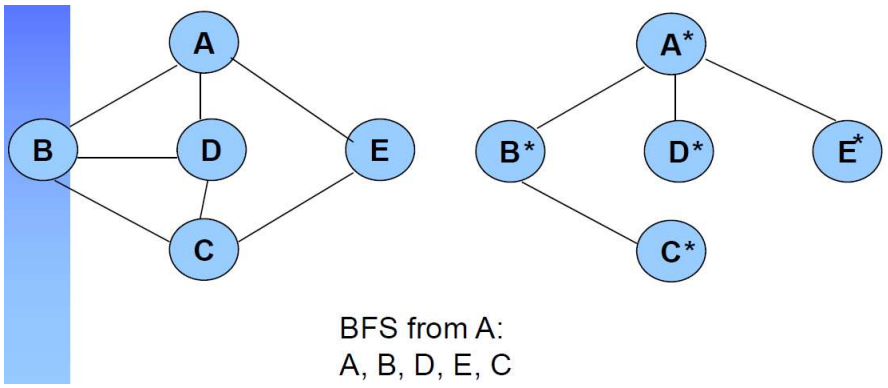
Analysis of DFS

- If G is represented by its adjacency lists, the DFS time complexity is $O(e)$.
  - There are $2e$ list nodes in the adjacency lists
- If G is represented by its adjacency matrix, then the time complexity to complete DFS is $O(n^2)$.

Breadth-First Search

- Breadth-First search (BFS): level order tree traversal
- BFS algorithm: using queue
- To avoid cycles, mark visited vertices
- If G is represented by its adjacency lists, the BFS time complexity is $O(e)$.
- If G is represented by its adjacency matrix, then the time complexity to complete BFS is $O(n^2)$.

Breadth-First Search



BFS from A:
A, B, D, E, C

Find Connected Components

- If G is an undirected graph, its connected components can be determined by calling DFS or BFS
- Check if there is any unvisited vertex
- Program 6.3 (pp.344) (next slide)

```
procedure COMP(G,n)
//connected components of G. G has n≧1 vertices.
VISITED is now a local array.//
for i = 1 to n do
  VISITED(i) = 0 //mark all vertices as unvisited//
end
for i = 1 to n do
  if VISITED(i) == 0 then
    call DFS(i); //find a component//
    output all newly visited vertices together
    with all edges incident to them
  end
end COMP
```

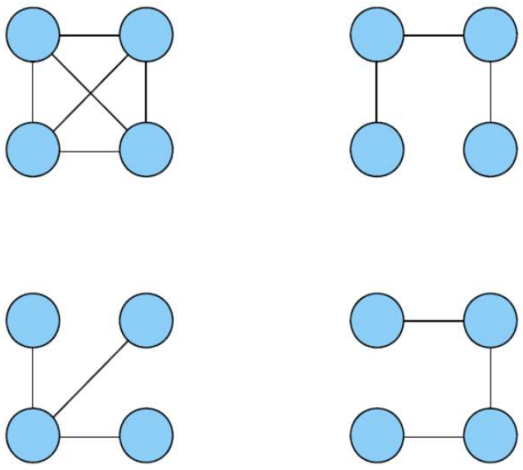

Connected Components (contd.)

- If G is represented by adjacency lists, the time complexity is $O(n+e)$
  - O(e) for DFS
  - O(n) for for loops
- If G is represented by adjacency matrix, the time complexity is $O(n^2)$

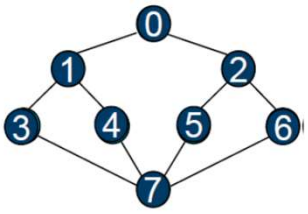
Spanning Trees

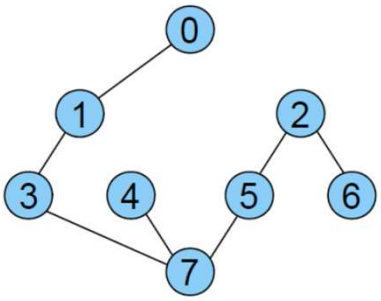
- Any tree consisting solely of edges in G and including **ALL** vertices in G is called a *spanning tree*.
  - Can be obtained by using either DFS or BFS.
- A spanning tree is a minimal subgraph, G', of G such that V(G') = V(G), and G' is connected. (Minimal subgraph is defined as one with the fewest number of edges).
- Any connected graph with n vertices must have at least *n-1* edges, and all connected graphs with *n–1* edges are trees. Therefore, a spanning tree has *n–1* edges.

A Complete Graph and Its Spanning Trees

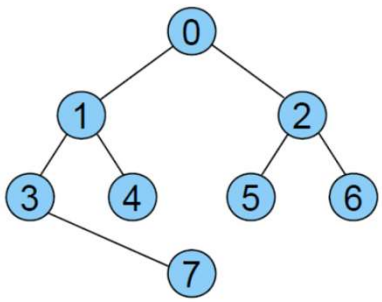


Depth-First and Breadth-First Spanning Trees





(a) DFS (0) spanning tree

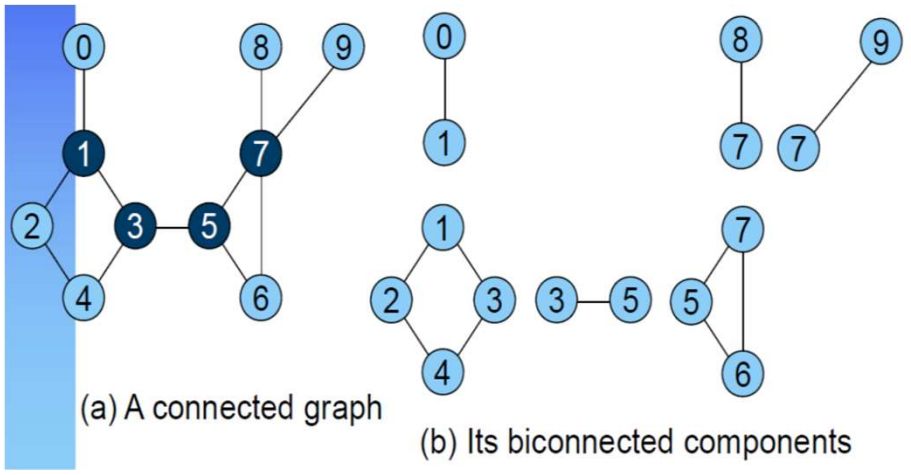


(b) BFS (0) spanning tree

Biconnected Components

- A vertex v of G is an articulation point iff the deletion of v, together with the deletion of all edges incident to v, leaves behind a graph that has at least two connected components.
  - e.g., vertices 1, 3, 5, 7 of Figure 6.20(a) (next slide)
- A biconnected graph is a connected graph that has **no** articulation points.
- A biconnected component of a connected graph G is a maximal biconnected subgraph H of G.
  - G contains no other subgraph that is both biconnected and properly contains H.

A Connected Graph and Its Biconnected Components



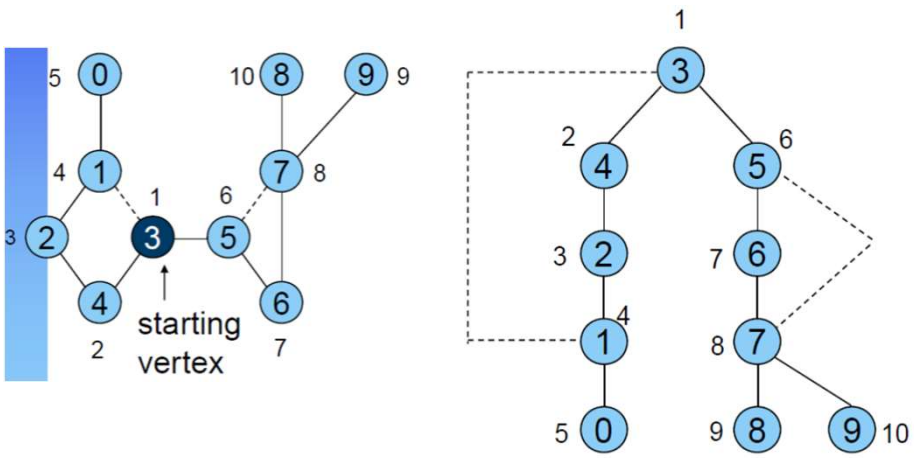
Biconnected Components

- Two biconnected components of the same graph can have at most one vertex in common.
- The biconnected components of G partition the edges of G.
  - No edge can be in two or more biconnected components (why?)
- The biconnected components of a connected, undirected graph G can be found by using any depth-first spanning tree of G.

Biconnected Components

- Edge (u,v) is a tree edge if v was first discovered by exploring edge (u,v)
- A nontree edge (u, v) is a back edge with respect to a spanning tree T iff either u is an ancestor of v or v is an ancestor of u
- A nontree edge that is not a back edge is called a cross edge.
- In a DFS of an undirected graph G, every edge of G is either a tree edge or a back edge (why ?)

Tree Edge & Back Edge in a DFS Tree

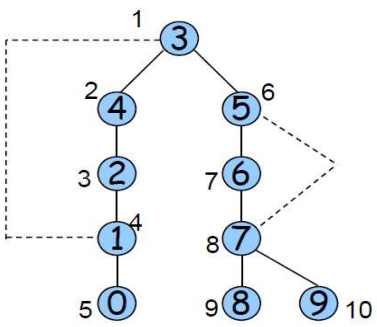


Biconnected Components (contd.)

- The root of the depth-first spanning tree is an articulation point iff it has at least two children.
- ***dfn(w)*** is defined as the order that w is discovered by DFS.
- Define ***low(w)*** as the lowest depth-first number that can be reached from *w* using a path of descendants followed by, at most, one back edge.
- *low*(*w*) = min{*dfn*(*w*), min{*low*(*x*) | *x* is a child of *w*}, min{*dfn(x)* | (w, x) is a back edge}}

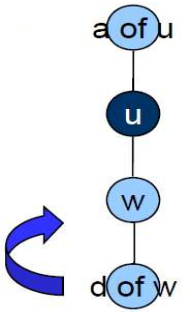
dfn and low Values for the Spanning Tree

vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	9	10
low	5	1	1	1	1	6	6	6	9	10



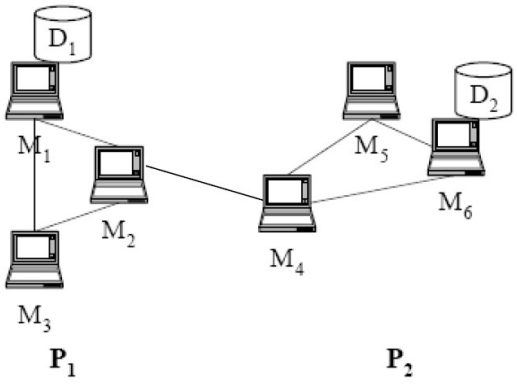
Biconnected Components (contd.)

- A vertex *u* is an articulation point iff
  - *u* is either the root of the spanning tree and has two or more children, or
  - *u* is not the root and *u* has a child *w* such that *low(w) ≥ dfn(u)*.



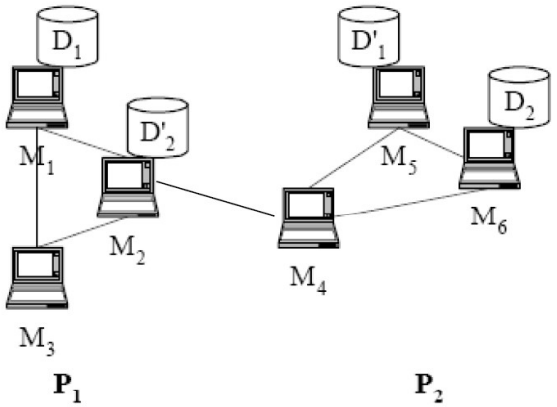
One Usage of Biconnected Components

- Ad-hoc network



(a) Without replication

One Usage of Biconnected Components (contd.)



(b) With replication

Minimum Cost Spanning Tree

- A *minimum-cost spanning tree* is a spanning tree of least cost
  - **Cost**: the sum of the costs (weights) of the edges in the spanning tree
- Three greedy-method algorithms available to obtain a minimum-cost spanning tree
  - **Kruskal's algorithm**
  - **Prim's algorithm**
  - **Sollin's algorithm**

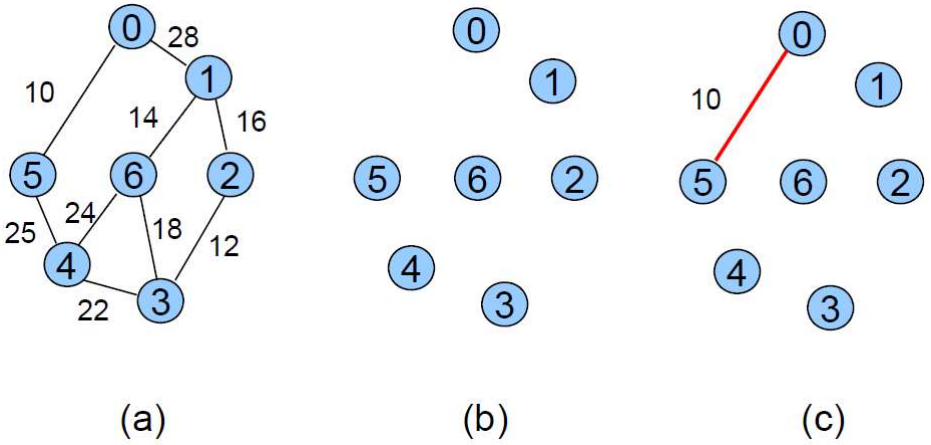
Minimal Cost Spanning Tree (contd.)

- Constraints
  - Must use only edges within the graph.
  - Must use exactly *n-1* edges.
  - May not use edges that produce a cycle.

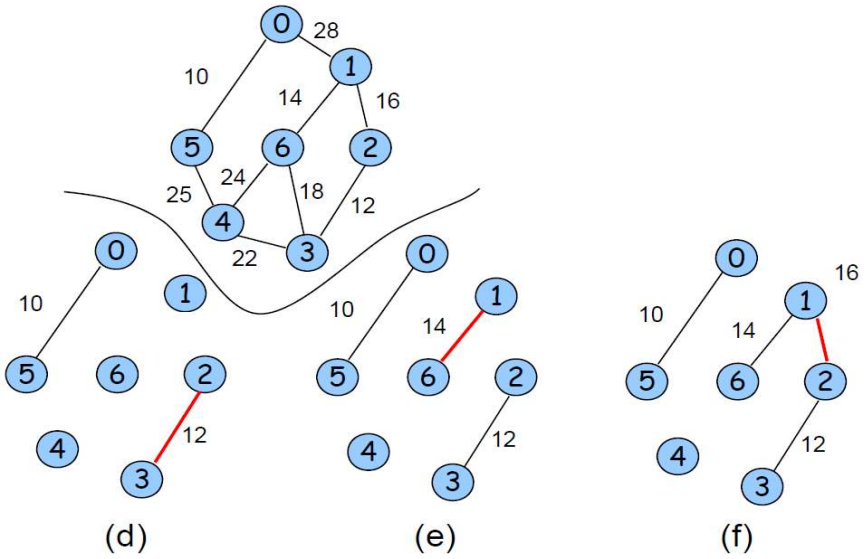
Kruskal's Algorithm

- Kruskal's algorithm builds a minimum-cost spanning tree T by adding edges to T one at a time.
- The algorithm selects the edges for inclusion in T in non-decreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges that are already in T.
- Theorem 6.1 (Kruskal's algorithm is correct)
- Time complexity: *O(e log e)*

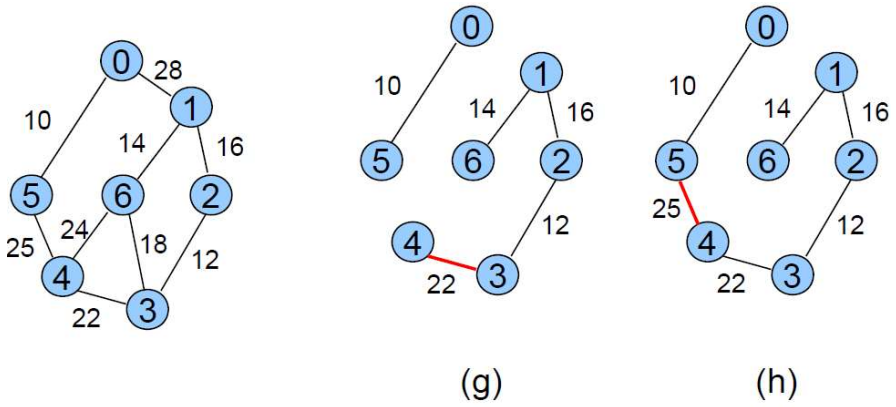
Stages in Kruskal's Algorithm



Stages in Kruskal's Algorithm (Cont.)



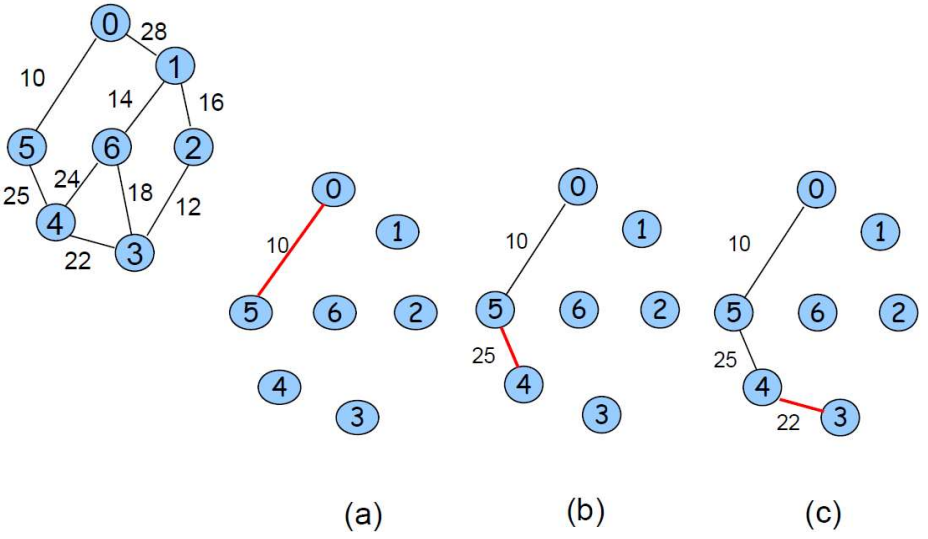
Stages in Kruskal's Algorithm (Cont.)



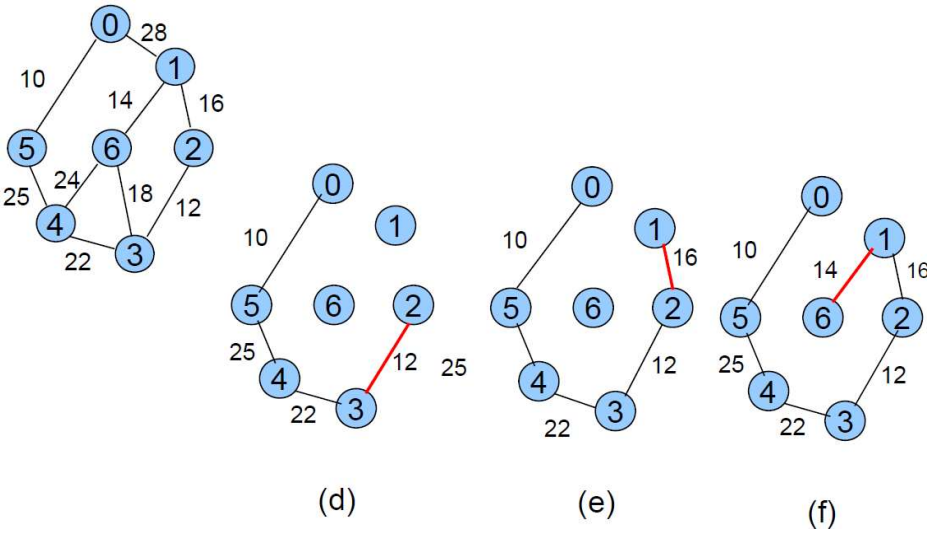
Prim's Algorithm

- The set of selected edges forms a tree at all times when using Prim's algorithm
  - In Prim's algorithm, a least-cost edge *(u, v)* is added to *T* such that $T \cup \{(u, v)\}$ is also a tree. This repeats until *T* contains *n-1* edges.
- Time complexity
  - $O(n^2)$
  - A faster implementation is possible when Fibonacci heap is used

Stages in Prim's Algorithm



Stages in Prim's Algorithm (Cont.)



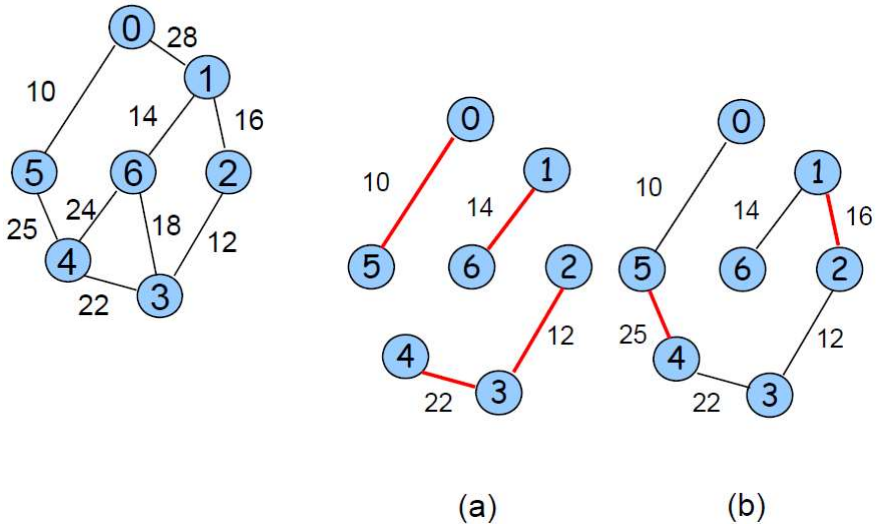
Sollin's Algorithm

- Contrast to Kruskal's and Prim's algorithms, Sollin's algorithm selects multiple edges at each stage
- At the beginning, all the n vertices form a spanning forest
- During each stage, a minimum-cost edge is selected for each tree in the forest.
  - The edges selected by vertices 0, 1, …, 6 are, respectively, (0,5), (1,6), (2,3), (3,2), (4,3), (5,0)

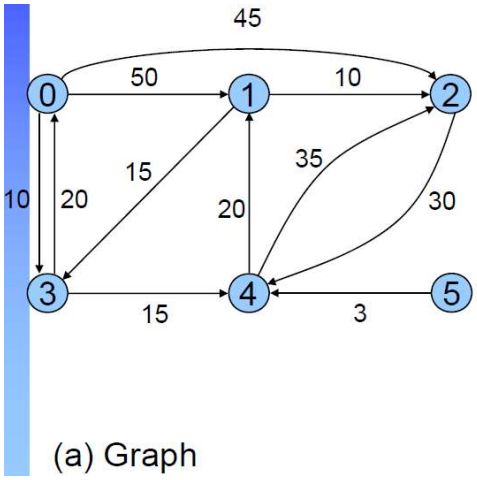
Sollin's Algorithm (contd.)

- It's possible that two trees in the forest to select the same edge. Only one should be used.
- Also, it's possible that the graph has multiple edges with the same cost. So, two trees may select two different edges that connect them together. Again, only one should be retained.

Stages in Sollin's Algorithm



Graph and Shortest Paths From Vertex 0 to All Destinations



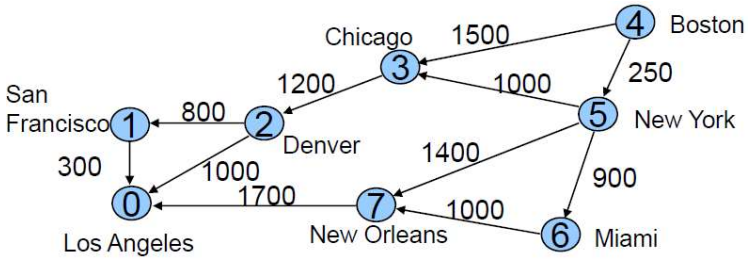
Path	Length
1) 0, 3	10
2) 0, 3, 4	25
3) 0, 3, 4, 1	45
4) 0, 2	45

(b) Shortest paths from 0

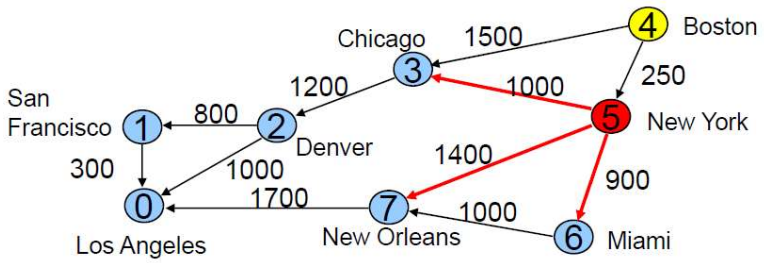
Single Source All Destinations: Nonnegative Edge Costs

- Let S denote the set of vertices (including v_0) to which the shortest paths have already been found.
  - If the next shortest path is to vertex *u*, then the path begins at v_0, ends at *u*, and **goes through only vertices that are in S**.
  - The destination of the next path generated must be the vertex *u* that has the **minimum distance among all vertices not in S**.
  - Select *u* to become a member of S.

Diagram for Example 6.5



	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0



Distance[3]=min{Distance[3],Distance[5]+length[5][3]}
=min{1500,250+1000}
=1250

Iteration	Vertex selected	S	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	4	{4}	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	5	{4,5}				1250				

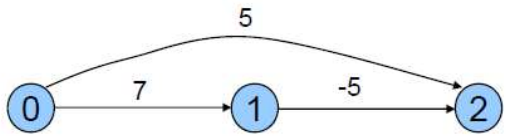
Action of Shortest Path

Iteration	Vertex selected	S	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	4	{4}	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	5	{4,5}	+∞	+∞	+∞	1250	0	250	1150	1650
2	6	{4,5,6}	+∞	+∞	+∞	1250	0	250	1150	1650
3	3	{4,5,6,3}	+∞	+∞	2450	1250	0	250	1150	1650
4	7	{4,5,6,3,7}	3350	+∞	2450	1250	0	250	1150	1650
5	2	{4,5,6,3,7,2}	3350	3250	2450	1250	0	250	1150	1650
6	1	{4,5,6,3,7,2,1}	3350	3250	2450	1250	0	250	1150	1650

Single Source All Destinations: Nonnegative Edge Costs (contd.)

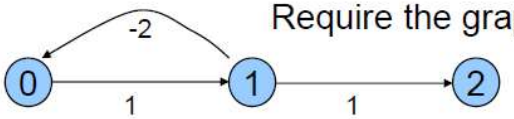
- The algorithm is first given by Edsger Dijkstra. Therefore, it's sometimes called Dijkstra Algorithm.
- Time complexity
  - Adjacency matrix, adjacency list: $O(n^2)$
  - Using Fibonacci heap: $O(n\log n+e)$

Directed Graphs



(a) Directed graph with a negative-length edge

Require the graph has no cycles



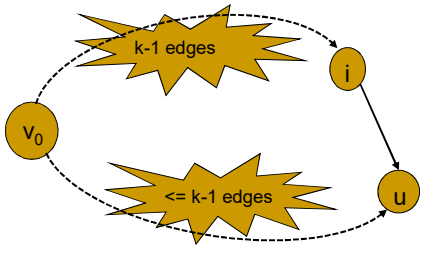
(b) Directed graph with a cycle of negative length

Single Source All Destinations: General Weights

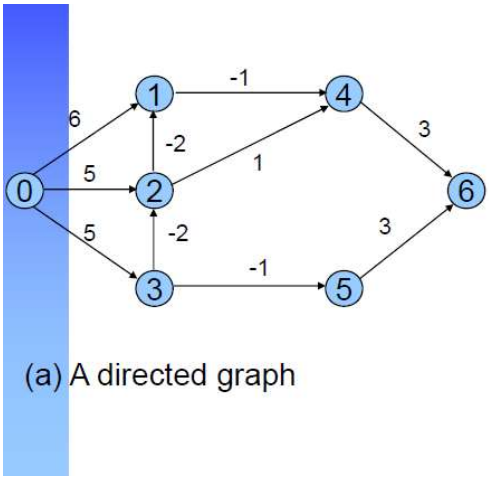
- When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most *n-1 edges on it.*
  - If the shortest path from v_0 *to u with at most k, k>1,* edges has no more than *k–1* edges, then
    $dist^k[u]=dist^{k-1}[u]$.
  - If the shortest path from v_0 to *u* with at most *k, k>1,* edges has exactly *k* edges, then it is comprised of a shortest path from v_0 to some vertex *i* followed by the edge <*i, u*>. The path from v_0 to *i* has *k–1* edges, and its length is $dist^{k-1}[i]$. (figure on next slide)

Single Source All Destinations: General Weights (contd.)

- The distance can be computed in recurrence by the following:
  $dist^k[u] = min\{ dist^{k-1}[u], min\{dist^{k-1}[i] + length[i][u]\} \}$
- The algorithm is also referred to as the Bellman-Ford Algorithm.
- Time complexity:
  - Adjacency matrix: $O(n^3)$
  - Adjacency list: $O(ne)$



Shortest Paths with Negative Edge Lengths



(a) A directed graph

k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2							
3							
4							
5							
6							

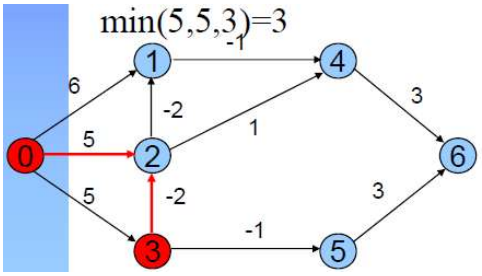
(b) $dist^k$

$dist^2[2]$

$dist^{k-1}[u]$
$dist^1[2]=5$

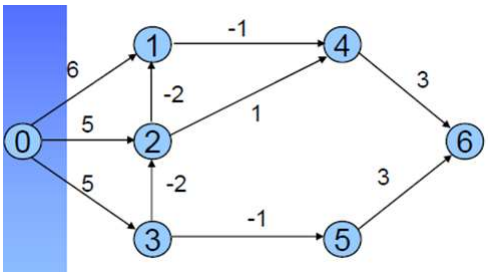
$dist^{k-1}[i]+length[i][u]$
$dist^1[0]+length[0][2]=5$
$dist^1[3]+length[3][2]=5-2$

min(5,5,3)=3



k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2		5	3	-2			
3							
4							
5							
6							

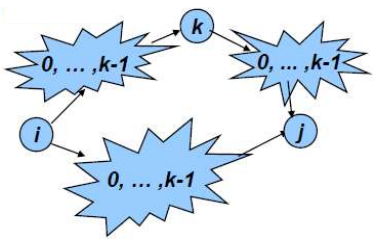
Shortest Paths with Negative Edge Lengths (contd.)



k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	**3**	**3**	5	5	4	∞
3	0	**1**	3	5	**2**	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

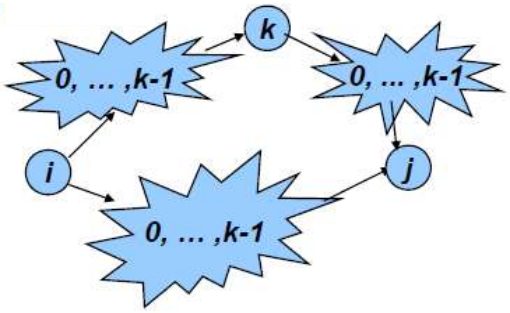
All-Pairs Shortest Paths

- Floyd-Warshall algorithm
- Notations
  - $A^{-1}[i][j]$: is just the *length[i][j]*
  - $A^{n-1}[i][j]$: the length of the shortest i-to-j path in G
  - $A^k[i][j]$: the length of the shortest path from i to j going through **no intermediate vertex of index greater than k**.

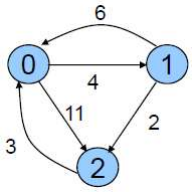


All-Pairs Shortest Paths (contd.)

- How to determine the value of $A^k[i][j]$
  - $A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$
- Time complexity
  - $O(n^3)$



Example for All-Pairs Shortest-Paths Problem



$A^0[2][1]$

$A^{-1}[2][1] = \infty$
$A^{-1}[2][0]+A^{-1}[0][1]=3+4=7$
$\min\{\infty,7\}=7$

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(a) A^{-1}

A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

(b) A^0

A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

(c) A^1

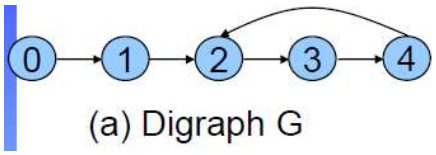
A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

(d) A^2

Transitive Closure

- **Definition**: The transitive closure matrix, denoted **A^+**, of a graph G, is a matrix such that $A^+[i][j] = 1$ if there is a path of length **>** 0 from i to j; otherwise, $A^+[i][j] = 0$.
- **Definition**: The reflexive transitive closure matrix, denoted **A^***, of a graph G, is a matrix such that $A^*[i][j] = 1$ if there is a path of length **>=** 0 from i to j; otherwise, $A^*[i][j] = 0$.

Graph G and Its Adjacency Matrix A, A^+, A^*



(a) Digraph G

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

(b) Adjacency matrix A

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

(c) A^+

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

(d) A^*

Activity-on-Vertex (AOV) Networks

- Definition: Activity-On-Vertex network or AOV network
  - A directed graph G
  - the vertices represent tasks or activities
  - the edges represent precedence relations between tasks.
- Definition: Vertex *i* in an AOV network G is a predecessor of vertex *j* iff there is a directed path from vertex *i* to vertex *j*.
  - *i* is an *immediate predecessor* of *j* iff <*i*, *j*> is an edge in G.
  - If *i* is a predecessor of *j*, then *j* is a *successor* of *i.*
  - If *i* is an immediate predecessor of *j,* then *j* is an *immediate successor* of *i.*

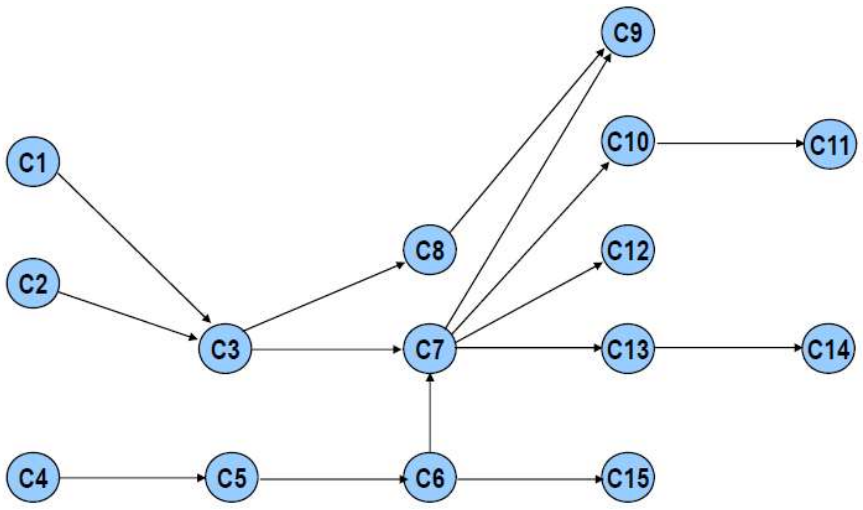
AOV Networks (contd.)

- Definition: A *topological order* is a linear ordering of the vertices of a graph such that, for any two vertices *i* and *j*, if *i* is a predecessor of *j* in the network, then *i* precedes *j* in the linear ordering. (i.e. a linear order which is consistent with all precedence relationships.)

An AOV Network of Courses

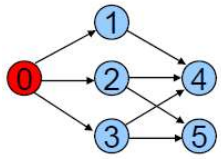
Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

An AOV Network (Fig 6.36)

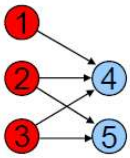


Action of Program 6.11(Topological Sorting) on an AOV Network(Fig 6.37)

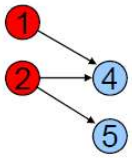
Pick a vertex v that has no predecessors (i.e., in-degree=0)



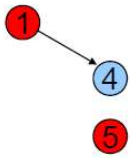
(a) Initial

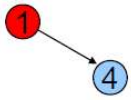


(b) Vertex 0 deleted



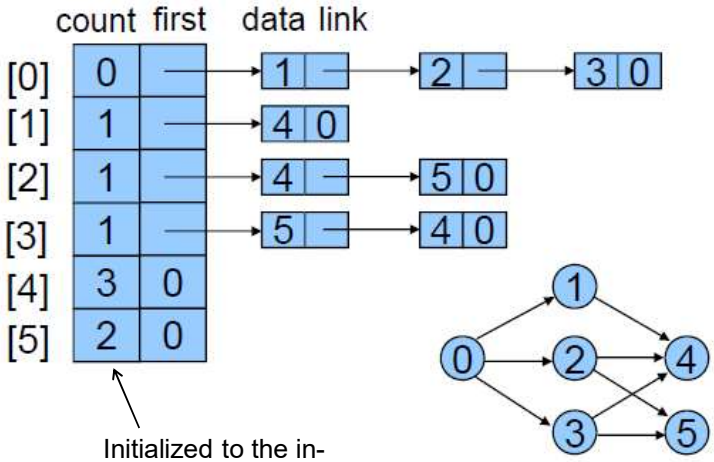
(c) Vertex 3 deleted





(d) Vertex 2 deleted (e) Vertex 5 deleted (f) Vertex 1 deleted

Using Adjacency List



An AOE Network

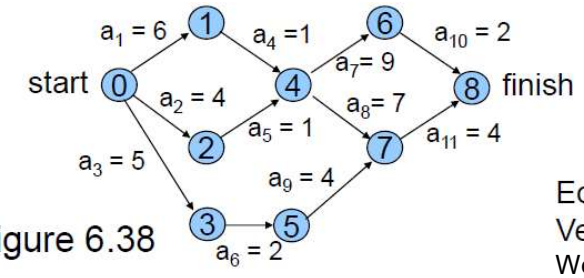


Figure 6.38

Edge: activity
Vertex: event
Weight: duration

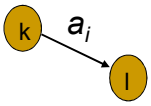
event	interpretation
0	Start of project
1	Completion of activity a_1
4	Completion of activities a_4 and a_5
7	Completion of activities a_8 and a_9
8	Completion of project

AOE Network

- A path of the longest length is a *critical path*
- The *earliest time* that an event *i* can occur is the length of the longest path from the start vertex 0 to the vertex *i*
- The earliest time an event can occur determines the *earliest start time* for all activities represented by edges leaving vertex *i*. (denoted by *e(i)*)
- For every activity a_i, let the latest time that an activity may start without increasing the project duration be *l(i)*.

AOE Network (contd.)

- All activities for which *e(i)=l(i)* are called *critical activities*
- For event *j* :
  - Earliest event occurrence time: *ee[j]*
  - Latest event occurrence time: *le[j]*
- If activity a_i is represented by edge <*k,l*>, we can compute:
  - *e(i)* = *ee[k]*
  - *l(i)* = *le[l]* - duration of activity a_i



AOE Network (contd.)

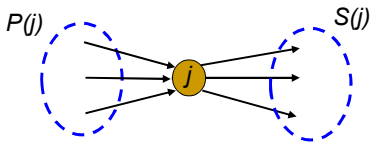
- Calculation of *ee[j]* and *le[j]*
  - *P(j)* is the set of all vertices adjacent to vertex *j*
  - *S(j)* is the set of all vertices adjacent from vertex *j*

$$ee[j] = \max_{i \in P(j)} \{ee[i] + \text{duration of} < i, j >\}, \text{ where}$$

$$ee[0] = 0$$

$$le[j] = \min_{i \in S(j)} \{le[i] - \text{duration of} < j, i >\}, \text{ where}$$

$$le[n-1] = ee[n-1]$$



- Using topological order

Adjacency Lists for Figure 6.38 (a)

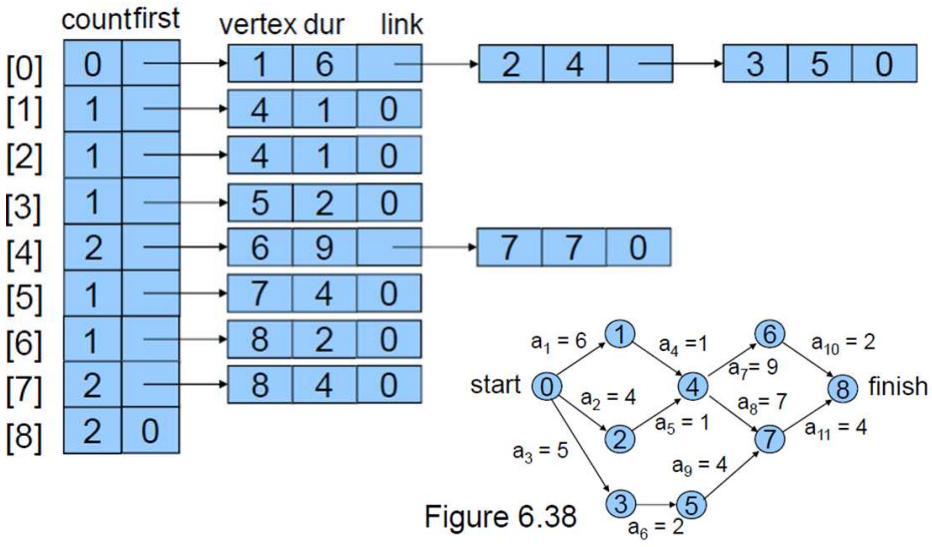


Figure 6.38

Adjacency Lists for Figure 6.38 (a)

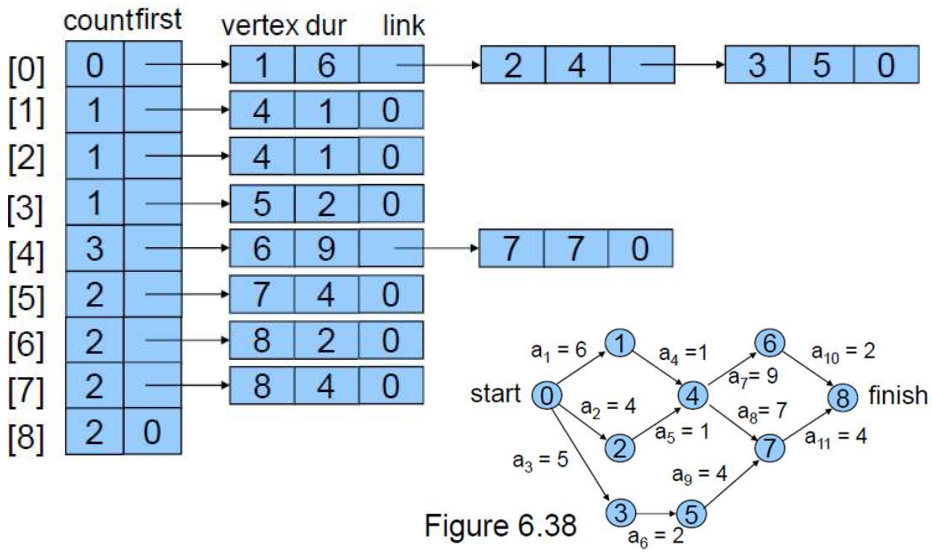


Figure 6.38

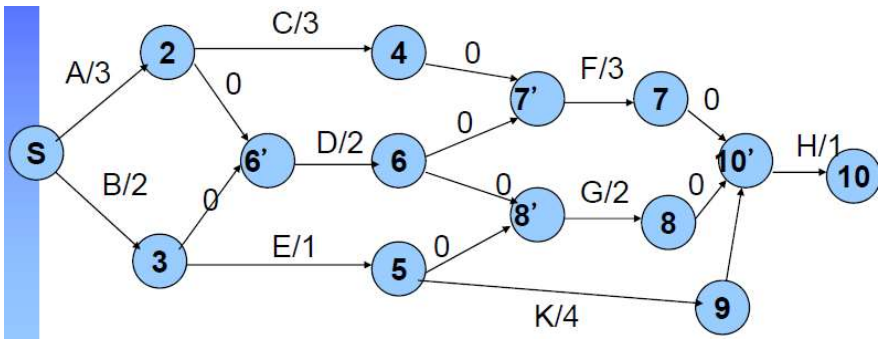
Computation of *ee*

In topological sorting, the vertices with in-degree=0 are placed in stack

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Initial	0	0	0	0	0	0	0	0	0	[0]
output 0	0	6	4	5	0	0	0	0	0	[3,2,1]
output 3	0	6	4	5	0	7	0	0	0	[5,2,1]
output 5	0	6	4	5	0	7	0	11	0	[2,1]
output 2	0	6	4	5	5	7	0	11	0	[1]
output 1	0	6	4	5	7	7	0	11	0	[4]
output 4	0	6	4	5	7	7	16	14	0	[7,6]
output 7	0	6	4	5	7	7	16	14	18	[6]
output 6	0	6	4	5	7	7	16	14	18	[8]
output 8										

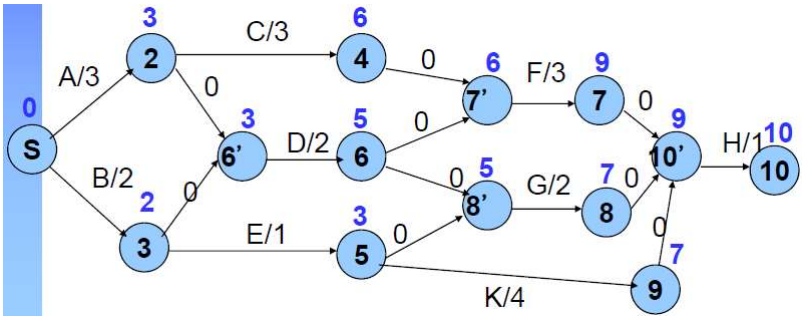
Critical Path Analysis

- AOE graph



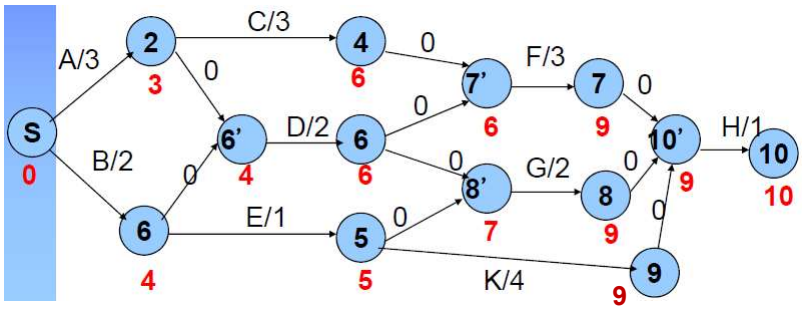
Critical Path Analysis (cont.)

- Earliest completion times: longest path
  - computed by topological order
  - $EE_1=0$
  - $EE_w=\max(EE_v+D_{v,w})$



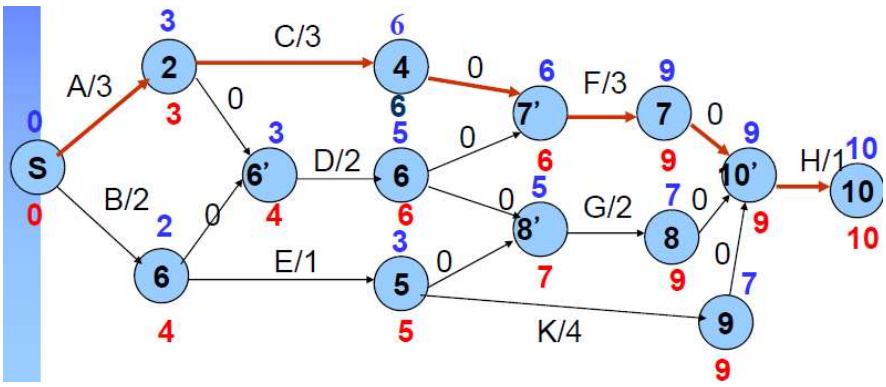
Critical Path Analysis (cont.)

- Latest completion times:
  - latest time without affecting final completion time
  - computed by reverse topological order
  - $LE_{10}=EE_{10}$
  - $LE_v=\min(LE_w - D_{v,w})$



Critical Path Analysis (cont.)

- Slack time(w) = LE_w - EE_w
- Critical path = zero slack time



Social Network Analysis (SNA)

- Social network analysis (SNA) is to discover patterns underlying social network.
- Replies on networks and graph theory.
- It can be used to analyze the social structure, relationships, interactions, … even the roles of actors(nodes) in the network.
- SNA has emerged as a key technique in modern sociology, economics, communication studies, organizational studies, …
- Even in fighting terrorism!

SNA – Centrality Measures

- To analyze which nodes(actors) are more important (or most important) in the network.
- Degree centrality – # edges connected to a node
- Closeness centrality – measure how close a node is to other nodes.
- Betweenness centrality – measure the fraction of paths that connect all pairs and include the node.
- Eigenvector centrality – mearue not only the connection degree but also the quality of connections.

SNA – Centrality Measure